

Executable HybridUML Semantics.  
A Transformation Definition.

von Stefan Bisanz

Dissertation

zur Erlangung des Grades eines Doktors der  
Ingenieurwissenschaften  
– Dr.-Ing. –

Vorgelegt im Fachbereich 3 (Mathematik & Informatik)  
der Universität Bremen  
im September 2005

Datum des Promotionskolloquiums: 14.12.2005

Gutachter: Prof. Dr. Jan Peleska (Universität Bremen)  
Prof. Dr. Rolf Drechsler (Universität Bremen)

# Preface

This thesis contributes to the development of hybrid systems. For the particular specification formalism HybridUML, a formally defined transformation  $\Phi$  of HybridUML models into executable low-level models is defined. The resulting low-level models have a formal semantics, therefore by  $\Phi$ , this formal semantics is assigned to the corresponding HybridUML models. As a result, there is no inconsistency between the HybridUML specification of a hybrid system and its implementation.

The work presented in this thesis has been investigated in the context of the HYBRIS (Efficient Specification of Hybrid Systems) project [DFG] supported by the Deutsche Forschungsgemeinschaft DFG as part of the priority programme on *Software Specification – Integration of Software Specification Techniques for Applications in Engineering*. The aim of this DFG priority programme has been the theoretically founded integration of different specification techniques and systematic proceedings for the development of safe software systems in complex applications in engineering. Different specification aspects have been taken into account, ranging from descriptions of physical systems to the different models of software systems. Mathematically founded specification techniques, as well as pragmatic ones which are used in practice of software production, have been considered.

The contribution of this thesis seamlessly integrates to the priority programme by the formal semantics definition for the specification formalism HybridUML. HybridUML is derived from the Unified Modeling Language, which is a wide-spread modeling language for the development of software systems. The semantics is executable, i.e. executable code is automatically generated from HybridUML models, such that they are directly usable in practice.

Chapter 1 motivates the use of the specification formalism HybridUML for the formal definition of hybrid systems. As an approach to combine the definition of hybrid systems in a formal, but user-friendly way, with the generation of a resulting executable system that has formally defined behavior, a transformation concept is proposed: Hybrid systems models are modeled with the specification language HybridUML, and are transformed into programs of the Hybrid Low-Level Framework HL<sup>3</sup>, which provides a restricted design pattern that the transformation has to comply with, as well as a runtime environment that provides basic functionality.

Chapter 2 defines the HybridUML Mathematical Meta-Model, which is a non-graphical definition of the HybridUML syntax. The separation of the meta-model from its graphical representation is the usual UML approach, and the benefits are that (1) the meta-model is directly usable for transformation  $\Phi$ , and that (2) the HybridUML semantics is independent from the graphical no-

tation. In chapter 3, the expressions that can be used within a HybridUML model, e.g. boolean expressions from mode invariants or transition conditions, or assignment expressions from transition actions, are defined by means of the HybridUML Expression Language.

The HL<sup>3</sup> Low-Level Framework is discussed in chapter 4. It is a compilation target for hybrid systems specification formalisms. A formal operational semantics is given for the execution of HL<sup>3</sup> models, which are defined as a mixture of explicit program code and abstractions to mathematical representations.

The specific transformation  $\Phi_{HUML}$  from HybridUML models to instances of the HL<sup>3</sup> framework is presented in chapter 5. The transformation is defined formally, therefore the HybridUML executable semantics results.

The thesis is concluded in chapter 6. A summary is given, and the main scientific contributions are pointed out. Possible future work related to HybridUML, to the low-level framework HL<sup>3</sup>, and to the transformational approach is discussed.

**Acknowledgements** I would like to thank everyone who has supported me in writing this thesis.

Sincere thanks to Prof. Dr. Jan Peleska, for contributing an extraordinary mixture of visionary leadership *and* detailed technical know-how, for motivating, and for providing all basic conditions that have facilitated this work.

Thanks to all current and former colleagues of the research group *Operating Systems and Distributed Systems*, in particular to Dr. Ulrich Hannemann, Jan Peleska, Kirsten Berkenkötter, and Aliko Ott for discussing and working on HybridUML and HL<sup>3</sup>.

Thanks a lot to my friends and family, for either not asking about HybridUML and HL<sup>3</sup>, or for their lack of understanding.

A special thanks to Aliko Ott, for her support in a variety of ways – for always producing excellent ideas, for discussing, for disagreeing, for asking uncomfortable questions, for motivating, and for being a very good friend.

Finally, a very special thanks to my boys Rasmus and Oskar, for reminding me of life's basics, and to my wife Kirsten, for being *really* patient.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	A Transformational Approach . . . . .	6
1.2	Related Work . . . . .	10
1.3	HybridUML by Example . . . . .	13
1.3.1	Radio-Based Train Control . . . . .	13
1.3.2	HybridUML model: Radio-Based Train Control . . . . .	15
<b>2</b>	<b>HybridUML Mathematical Meta-Model</b>	<b>29</b>
2.1	Structural Specification . . . . .	31
2.2	Behavioral Specification . . . . .	38
<b>3</b>	<b>HybridUML Expression Language</b>	<b>43</b>
3.1	Context of Expressions . . . . .	45
3.2	Identifier Expressions . . . . .	49
3.2.1	Syntax of Identifier Expressions . . . . .	50
3.2.2	Intermediate Semantics of Identifier Expressions . . . . .	51
3.3	HybEL Expressions . . . . .	53
3.3.1	Syntax of HybEL Expressions . . . . .	54
3.3.2	Intermediate Semantics of HybEL Expressions . . . . .	61
3.4	Skeleton Evaluation Semantics . . . . .	66
<b>4</b>	<b>HL<sup>3</sup> – Hybrid Low-Level Framework</b>	<b>69</b>
4.1	HL <sup>3</sup> Overview . . . . .	70
4.1.1	Runtime Environment . . . . .	70
4.1.2	Design Pattern . . . . .	72
4.2	CONST – Constant State Components . . . . .	77
4.2.1	Entities . . . . .	77
4.2.2	Dependencies . . . . .	79
4.2.3	Physical Constraints . . . . .	81
4.3	VAR – Variable State Components . . . . .	81
4.4	Scheduling Rules . . . . .	89
4.4.1	Timing . . . . .	93
4.4.2	Switching between Execution Phases . . . . .	98
4.4.3	Scheduling within Execution Phases . . . . .	101
4.5	Abstract Subject Execution . . . . .	103
4.5.1	Progress of Abstract Subject Execution . . . . .	104
4.5.2	Termination of Abstract Subject Execution . . . . .	105
4.6	Program Subject Execution . . . . .	110

4.6.1	Progress of Statement Execution . . . . .	111
4.6.2	Termination of Statement Execution . . . . .	112
4.6.3	Program Termination . . . . .	117
<b>5</b>	<b>Transformation Definition</b>	<b>119</b>
5.1	Intermediate Specification Representation . . . . .	120
5.1.1	Structure . . . . .	121
5.1.2	Behavior . . . . .	129
5.2	Evaluation Semantics of HybEL Expressions . . . . .	134
5.3	Prerequisites for Code Creation . . . . .	136
5.3.1	Variables and Signals of Expressions . . . . .	136
5.3.2	Variable and Signal Nodes of Expression Nodes . . . . .	140
5.3.3	Hybel Item Trees of Expression Nodes . . . . .	140
5.4	HL <sup>3</sup> Model Definition . . . . .	141
5.4.1	Entities . . . . .	141
5.4.2	Dependencies . . . . .	149
5.4.3	Physical Constraints . . . . .	153
5.5	Code Creation for Expression Nodes . . . . .	154
5.5.1	Evaluation Interpretation of Hybel Item Trees . . . . .	158
5.5.2	Evaluation Interpretation of Identifier Item Trees . . . . .	164
5.5.3	Assignment Interpretation of Hybel Item Trees . . . . .	166
5.5.4	Assignment Interpretation of Identifier Item Trees . . . . .	170
5.5.5	Initialization of Local Variables from Channels . . . . .	173
5.5.6	Visibility Set Parameter Access . . . . .	175
5.5.7	Publication of Local Variable Data to Channels . . . . .	176
5.6	HybridUML Abstract Subject Execution . . . . .	180
5.6.1	Abstract Machines . . . . .	180
5.6.2	HybridUML Selector . . . . .	188
<b>6</b>	<b>Conclusion</b>	<b>195</b>
6.1	Summary . . . . .	195
6.2	Contributions . . . . .	196
6.3	Future Work . . . . .	197
6.3.1	Enhancements . . . . .	197
6.3.2	Further Investigations . . . . .	198
<b>A</b>	<b>Mathematical Notations</b>	<b>201</b>
A.1	Sets of Standard Values . . . . .	201
A.2	Power Sets . . . . .	201
A.3	Cardinality of Sets . . . . .	201
A.4	Element Selection . . . . .	201
A.5	Functions . . . . .	202
A.5.1	Function Notations . . . . .	202
A.5.2	Domain and Range of Functions . . . . .	202
A.5.3	Inverse of a Function . . . . .	202
A.5.4	Overriding of Functions . . . . .	202
A.6	Projection Functions . . . . .	202
A.7	Sequences . . . . .	203
A.8	Trees . . . . .	203
A.8.1	Unordered Trees . . . . .	203

A.8.2	Ordered Trees . . . . .	204
<b>B</b>	<b>HybEL Grammar</b>	<b>205</b>
B.1	Expressions . . . . .	205
B.2	Boolean Expressions . . . . .	205
B.3	Numeric Expressions . . . . .	206
B.4	Enumeration-typed Expressions . . . . .	206
B.5	Structured Data Type Expressions . . . . .	206
B.6	Differential Expressions . . . . .	207
B.7	Boolean Operations . . . . .	207
B.8	Numeric Operations . . . . .	207
B.9	Integer Set Expression . . . . .	208
B.10	Assignment Expressions . . . . .	208
B.11	Signal Raise Statements . . . . .	209
B.12	Trigger Expressions . . . . .	209
B.13	Variables . . . . .	210
B.14	Signals . . . . .	211
B.15	Literals . . . . .	211
<b>C</b>	<b>Case Study: Radio-Based Train Control</b>	<b>213</b>
C.1	Case Study Description . . . . .	213
C.2	HybridUML model: Radio-Based Train Control . . . . .	216
C.2.1	System . . . . .	216
C.2.2	Data Types . . . . .	219
C.2.3	Train . . . . .	224
C.2.4	User . . . . .	225
C.2.5	LocalizationDevice . . . . .	226
C.2.6	Movement . . . . .	227
C.2.7	Brake . . . . .	228
C.2.8	Engine . . . . .	228
C.2.9	TrainController . . . . .	229
C.2.10	CloseRequestController . . . . .	231
C.2.11	CrossingStatusController . . . . .	236
C.2.12	TrainRadioController . . . . .	243
C.2.13	UserInteractionController . . . . .	244
C.2.14	LocalizationController . . . . .	244
C.2.15	MovementController . . . . .	246
C.2.16	EmergencyController . . . . .	248
C.2.17	BrakePointController . . . . .	250
C.2.18	Crossing . . . . .	254
C.2.19	Lights . . . . .	256
C.2.20	DangerZone . . . . .	258
C.2.21	LogicalGateBarrier . . . . .	259
C.2.22	SwitchOffSensor . . . . .	260
C.2.23	Gate . . . . .	261
C.2.24	CrossingController . . . . .	265
C.2.25	LightController . . . . .	267
C.2.26	GateController . . . . .	268
C.2.27	SwitchOffSensorController . . . . .	270
C.2.28	DefectWatcher . . . . .	271

C.2.29	CrossingRadioController . . . . .	273
C.2.30	ProtocolController . . . . .	274
C.2.31	OperationsCenter . . . . .	278
C.2.32	OcEnvironment . . . . .	279
C.2.33	OcController . . . . .	281
C.2.34	OcCoreController . . . . .	282
C.2.35	OcRadioController . . . . .	284
C.2.36	RadioChannelTrainCrossing . . . . .	285
C.2.37	RadioChannelTrainOc . . . . .	286
C.2.38	RadioChannelCrossingOc . . . . .	287
<b>D</b>	<b>Code Examples</b>	<b>289</b>
D.1	HL <sup>3</sup> Model . . . . .	290
D.1.1	Definition of Channels and Ports . . . . .	290
D.1.2	Definition of Abstract Machines . . . . .	292
D.1.3	Instantiation of Selector and Scheduler . . . . .	295
D.2	Created Code for Expression Nodes . . . . .	296
D.2.1	Flow::expression53 . . . . .	296
D.2.2	Flow::expression54 . . . . .	297
D.2.3	Flow::expression55 . . . . .	298
D.2.4	Flow::expression56 . . . . .	298
D.2.5	Flow::expression57 . . . . .	299
D.2.6	Flow::expression58 . . . . .	300
D.2.7	Action::trans17 . . . . .	301
D.2.8	Action::trans18 . . . . .	301
D.2.9	Action::trans19 . . . . .	301
D.2.10	Action::trans20 . . . . .	302
D.2.11	Action::trans21 . . . . .	303
D.2.12	InvariantConstraint::expression41 . . . . .	304
D.2.13	InvariantConstraint::expression42 . . . . .	306
D.2.14	InvariantConstraint::expression43 . . . . .	308
D.2.15	Guard::expression44 . . . . .	309
D.2.16	Guard::expression45 . . . . .	309
D.2.17	Guard::expression46 . . . . .	309
D.2.18	Guard::expression48 . . . . .	311
D.2.19	Guard::expression51 . . . . .	312
	<b>Bibliography</b>	<b>315</b>

# Chapter 1

## Introduction

This thesis contributes to the development of hybrid systems. For the particular specification formalism HybridUML, a formally defined transformation  $\Phi$  of HybridUML models into executable low-level models is defined. The resulting low-level models have a formal semantics, therefore by  $\Phi$ , this formal semantics is assigned to the corresponding HybridUML models. As a result, there is no inconsistency between the HybridUML specification of a hybrid system and its implementation.

In the first chapter, the topic of hybrid systems and hybrid systems modeling is introduced. The specific formalism HybridUML is presented by example, modeling a case study from the railway domain. The transformation concept is motivated, combining three major benefits for the development of hybrid systems: (1) The specification formalism is usable, taking advantage of the well-accepted Unified Modeling Language. (2) From HybridUML models, executable code is generated automatically. (3) HybridUML models have a formal semantics.

The details of the transformation  $\Phi$ , along with the definition of a framework for low-level models, is given in the subsequent chapters.

Many people are affected by a lot of computer systems today. At least in the so-called “developed countries” of this earth, computers are almost everywhere. Even if someone deliberately tries to avoid computer systems, she will not succeed, because everyday life is accompanied by a variety of embedded computer systems: household appliances like cookers, microwave ovens, washing machines, home automation products like thermostats or air conditioners, entertainment devices like television sets are equipped with computers nowadays. Just leaving home will not help, either, because transportation without computers is rare, too: cars, trains, aircrafts have lots of embedded computers inside. Further, it won’t suffice to leave the cellular phone at home, along with the digital wristwatch, and to have a walk, because computer systems of someone else are around. The failure of the antilock brake controller of an approaching car, for example, could become relevant, and the benefit of medical equipment based on modern computer technology could be welcome.

While many computer systems have the potential to harm people, or the environment, they are beneficial in a variety of ways – they enhance safety, they provide comfort, they entertain, they probably increase productivity, and so on. Of course, this is the motivation to use computer systems. Unfortunately, the development of computer systems is a very complex task, and the software

crisis of the 1970's is everything but solved: it is still a time-consuming task to develop a computer system and the corresponding software in a way that the system behaves adequately, i.e. that it meets its requirements, and that it has the right requirements. In other words, it is *expensive*.

Like for most human concerns, the main measure for computer systems is money. That means, the trade-off between the development effort, the expenses on running the system, the (expected) benefit of the system, and the risk of system failures is a financial calculation. Even human lives are not invaluable, but have a price.

Therefore, to improve the quality of computer systems on the one hand, and to reduce the development costs on the other, has been a central issue for a long time, and is a central issue for today. The thesis at hand contributes to this in a specific way: for the *development of hybrid systems*, software quality is enhanced by an automatic transformation of high-level models into semantically well-defined code. No manual coding activity is needed for this, therefore costs are reduced, too.

**Hybrid Systems.** Computer systems are no self-contained systems, because then they would be useless. In fact, a computer system is always a part of an overall system. The largest known overall system is the universe, and every smaller system is contained in it. Usually, a smaller system can be identified that sufficiently defines the scope of the computer system, that is, its *operational environment*.

In order to emphasize the part-of relationship, most everyday-life computer systems are so-called *embedded computer systems*. Such a computer system is some kind of control system that interacts with its operational environment, monitoring some observables (e.g. temperature, speed) via sensors and setting others (e.g. voltage, thrust) using actuators. In contrast, the popular personal computer which people have in mind when thinking about computer systems, generally has a less tight coupling to its environment.

For the operation of embedded computer systems, time is a key issue. Not only the correctness of calculations is important, but also the points in time when they become effective. The operational environment – which is (some part of) the *physical environment* – inherently provides a notion of time. The overall system thus is a *real-time system*.

More formally, a real-time system is a system that changes its state as a function  $state_{sys} : Time \rightarrow State_{sys}$  of physical time [Kop97]. Many (solely) physical systems can be seen as *continuous real-time systems* [Sto96], i.e.  $state_{sys}$  is totally differentiable. This is what naturally happens when temperature increases in the sunlight, wind strength changes etc.

In contrast, computer systems are often modeled as *discrete real-time systems*, due to the binary nature of today's computer technology. The state function  $state_{sys}$  of discrete systems is partially constant, with non-differential points that define discrete state changes. That means, the state is assumed to evolve stepwise.

Traditionally, complete systems were modeled either as continuous systems, or as discrete systems. Therefore, either the description of the computer system, or the model of the operational environment often was inadequate. The explicit unification of both aspects leads to the notion of *hybrid systems*, which are real-

time systems with partially differentiable (but not necessarily constant)  $state_{sys}$ . A hybrid system is a system that can be modeled best as mixture of discrete and continuous state changes.

Roughly, the computer system can be seen as the discrete part of the system, and its operational environment as the continuous part. But for modeling the overall system, it is often convenient to avoid a strict separation. For HybridUML, we will allow continuous specifications for the computer system – also called *hybrid computer system* – which will be discretized by the transformation into the target code. The other way round, the environment may define discrete state changes, because it is not restricted to be purely physical. It can also contain computer systems itself, or human operators who trigger discrete state changes, by pressing a button, for example.

**Formal Specifications.** In order to define a *formal semantics* for a model of a hybrid system, a *formal specification* of the model is mandatory, and therefore a *formal specification language* is needed. Various specification languages exist for the modeling of hybrid systems; we point out two of them that we consider as essential, and that have acted as the starting point for the definition of HybridUML [BBHP03].

*Hybrid Automata.* This is a state-based formalism augmented by real-valued variables which may continuously evolve over time. A hybrid automaton contains a discrete control state and a continuous state. The control state space is defined by control modes<sup>1</sup> which are linked by control switches<sup>2</sup>, such that the activation of a control switch represents a discrete behavioral step. The continuous state space is given by a finite set of real-valued variables which are modified by flow conditions that are attached to control modes. Therefore, the current control state defines the evolution of the continuous states.

Invariant conditions affect the control state, in that they restrict control modes to given continuous sub-states, such that an associated control mode must be left, if the invariant condition is violated.

Jump conditions of control switches then determine the enabledness of a control switch. Additionally, the continuous state can be modified, therefore they represent conditions as well as actions of control switches.

The semantics of a hybrid automaton is given as a labeled transition system with an infinite number of states and transitions. Transitions either represent a discrete state change, or the flow of time, distinguished by transition labels. An infinite number of transitions is necessary to define the possible flows for time durations  $\delta \in [0, t_{up}]$ .

Hybrid Automata were introduced independently by [ACHH93] and [NOSY93], and later presented as joint work in [ACH<sup>+</sup>95]. An overview is given in [Hen96].

*CHARON.* The language CHARON [AGLS01, ADE<sup>+</sup>01, ADE<sup>+</sup>03] extends the concepts of Hybrid Automata, in that it allows the specification of architectural as well as behavioral hierarchy and supports prioritized behavior or

---

<sup>1</sup>For discrete automata representations, the usual term is *state*.

<sup>2</sup>A more common term is *transition*.

exception handling through so-called group transitions. It therefore facilitates the application to large-scale systems.

A CHARON specification is given by a set of *agents* which communicate with their environment via shared variables. Basically, an agent consists of a set of variables and a set of *modes*, where a mode is basically a hierarchical state machine containing (sub-)modes itself. Further, a mode contains a set of *transitions* that connect its submodes (or the mode itself with a submode) via *control points*, a set of variables, and a set of constraints.

Similarly to Hybrid Automata, either some transition is taken, where variables are changed instantaneously, or some time passes, where the system resides in the same mode and the continuous variables change over time. This is restricted by corresponding constraints, which are given as differential (in-)equations, as algebraic constraints, and as invariants, in a similar fashion as for Hybrid Automata, but they are evaluated hierarchically at the various levels to simplify the description of the system.

When a mode is executing a continuous step, i.e. when time passes, then the hierarchical state machine as a whole is acting. For time-delay steps, modes on all levels, from the top-level mode to the leaf modes, have to coordinate for this. Part of this coordination is that any possible valuation of analog variables has to comply to the constraints attached to all active modes on the various levels. The change of values over time for a mode is thus described in terms of the constraints of that mode and the relations associated with its submodes. For time passing steps it is obvious that all components in a concurrent hybrid system have to participate in such a step.

The language supports the operations of composition of agents to model concurrency, hiding of variables to restrict sharing of information, and instantiation of agents to support reuse.

The semantics of a CHARON model is given as an infinite-state labeled transition system, with pairs  $(c, \sigma)$  being the states, encoding the discrete state of the CHARON model with control point  $c$  and the continuous state as valuation  $\sigma$  of continuous variables. As for Hybrid Automata, the flow of time is represented by an infinite number of transitions, and is distinguished from discrete state changes by transition labels.

**UML.** Independently of the formal nature of specifications that we require, we are convinced that a specification language will only be used in practice, (1) when the syntax is accepted by its potential users, and (2) when there is satisfactory tool support available. Seemingly, today's specification formalisms for hybrid systems do not provide this.

In contrast, the Unified Modeling Language (UML) [RJB04, UML] is the canonical candidate to fulfill the above requirements, since it is the most widespread software-engineering formalism today. Therefore, various CASE tools exist which support UML, and many users are already familiar with UML.

For UML, a formal semantics is *not* defined, instead, the syntactic constructs of the language are only associated with informal meanings. Their semantic interpretation is deliberately left to the specific project context, to the application domain, and to the implementation technique. The precise behavior is only obtained by the applied transformation into the corresponding target programming language.

Probably, this is one of the reasons for UML's popularity, because for this the scope of the application of UML is rather unrestricted. However, the drawbacks of this approach are that (1) in general, it is infeasible to capture the behavior of software written in Java, C/C++, or Ada, when executed in a specific target environment, and that (2) therefore, the semantics of a particular UML model remains unexplained to a considerable extent. The situation is even worse, if the transformation is only given informally (e.g. if code creation is a manual activity) – then a particular UML model in a given context may have more than one semantics.

**Executable Models – Code Generation.** The practical usability of a formal specification language fundamentally depends on how easily it can be transformed into an executable target, i.e. program code. The additional effort which is invested into the formal specification of a model – in contrast to the effort spent on elaborating informal conventional specifications – is only worth while automatic code generation is available. As a counter-example, we do not expect that developing executable code by step-wise refinement is a promising approach. For automatic code generation, it is essential that the executable target's behavior corresponds to the semantics of the model, because that is the main benefit of the formal model. Only then, further efforts like verification or model checking of the formal model have a direct effect on the quality of the target.

For several specification formalisms for hybrid systems, tools are available for analysis or simulation. For example, for Hybrid Automata, there is a model checker called *HyTech* [HHWT97], and for CHARON, a (textual) editor with parser and type checker as well as a simulator exist [CHA]. But to our knowledge, there is no specification formalism for hybrid systems that provides the generation of optimized code to be used for embedded control systems.

As opposed to this, among the variety of tools that support the Unified Modeling Language, there are several ones that generate executable code from UML models, for example *Together* [Tog], *Rhapsody* [Rha], or *Rational Rose* [Rat]. As a direct consequence of the missing formal semantics of UML, the behavior of the generated code corresponds to some interpretation of the informal descriptions given with UML, which is different for all these tools.

**HybridUML.** In order to enhance the development of hybrid computer systems, we already have proposed HybridUML [BBHP03, BBHP04a] as a formal specification language which is syntactically based on UML. In [BBHP04a] we have described an approach to transform HybridUML specifications into targets of an executable low-level framework that has a formal semantics.

The transformational approach defines the semantics of HybridUML specifications and the executable target *at once*, and therefore eliminates the gap between the formal semantics of hybrid systems' models and the behavior of their implementations.

As a consequence, HybridUML facilitates the modeling of hybrid computer systems in a convenient way. Models of hybrid systems can be created based on the well-known specification language UML, using standard UML tools. From these models, executable code can be generated automatically, which has a defined semantics and is therefore not subject to ambiguities. HybridUML not

only aims to hybrid systems' designers and developers, but also to managers and customers, since UML's graphical nature is also suitable for informal discussions on requirements and designs.

This thesis provides the full definition of the transformation of HybridUML models into executable low-level models. In section 1.1 the transformational approach is presented in detail, and the outline of the thesis is given. Section 1.2 provides further references to related work.

## 1.1 Executable HybridUML Semantics: A Transformational Approach

The aim of the specification language HybridUML is to close the gap between (1) the need for a formal specification language that is suitable for hybrid systems modeling and (2) the facility to generate executable code that preserves the semantics of the specification. In order to guarantee sufficient tool support, and to ensure wide user acceptance, (3) UML as a well-accepted formalism of software engineering is chosen as a syntactic framework. Language extension is an inherent feature of UML, therefore well-constructed UML tools should support HybridUML as well.

**Transformational Semantics.** The semantics of HybridUML is defined by a transformation  $\Phi$  that maps valid HybridUML models into models of the *Hybrid Low-Level Framework* HL<sup>3</sup>.

HL<sup>3</sup> models are executable, because they consist of executable program code. For the presentation in this thesis, HL<sup>3</sup> models are defined as a mixture of explicit program code and abstractions to mathematical representations. On this basis, a formal operational semantics is given for HL<sup>3</sup> models.

The transformation  $\Phi$  also is defined formally, such that from a given HybridUML specification, a formally defined HL<sup>3</sup> *program* with a formal operational semantics results. Input for the transformation is a valid HybridUML model, which results from a combination of graphical and textual elements.

The concepts of the transformational approach have been proposed initially in [BBHP04a]. This thesis gives a *full definition* of the transformation  $\Phi$ , as well as of the underlying Hybrid Low-Level Framework HL<sup>3</sup>.

Figure 1.1 illustrates the transformational approach, its components are described in the following.

**HybridUML Syntax.** As usual for UML, the syntax of valid HybridUML models is given by a *meta-model* that separates the definition of model elements from their corresponding graphical notations. For the initial proposal of HybridUML [BBHP03], the HybridUML meta-model was defined in terms of a *UML profile*, which is the recommended approach to customize UML since version UML 2.0. A profile defines constraints and/or extensions on the UML meta-model and therefore defines the resulting language indirectly.

For the purpose of this thesis, chapter 2 provides an explicit mathematical definition of valid HybridUML models. This is done for two reasons: (1) The formal definition of the transformation  $\Phi$  is given mathematically and therefore

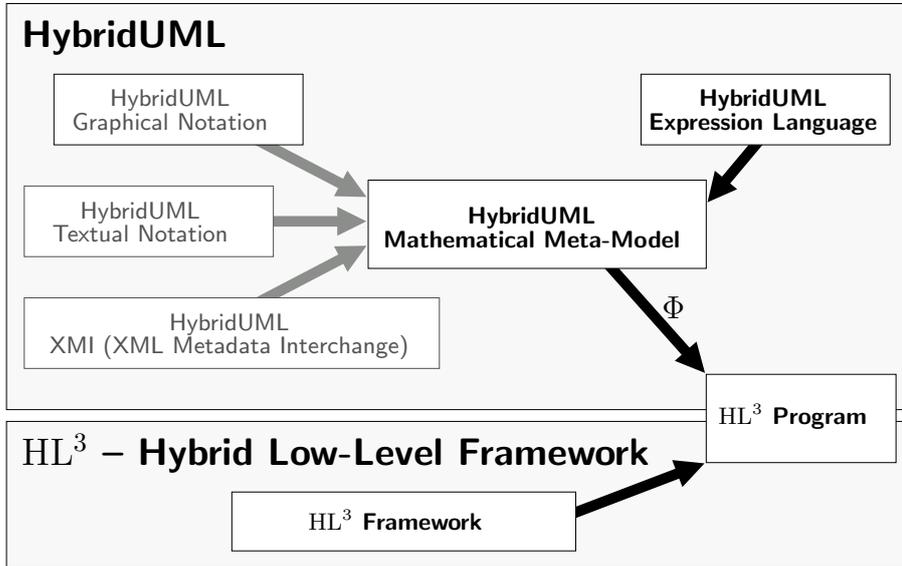


Figure 1.1: HybridUML Overview. The Meta-Model, the Expression Language, the  $HL^3$  Framework, and the transformation  $\Phi$  to  $HL^3$  Programs are defined in this thesis.

requires an adequate HybridUML representation. (2) HybridUML *restricts* the UML to a large extent, therefore an explicit language definition is far more compact than the formal restriction of the complete UML meta-model.

The definition of the graphical HybridUML notation is out of scope of this thesis. In practice, it will depend on the specific tool which is chosen to implement the mapping from graphical HybridUML specifications to HybridUML models. Conceptually, there are three different ways to realize this mapping: (1) An appropriate tool is extended such that it provides valid HybridUML models directly to the transformation  $\Phi$ . (2) A textual HybridUML notation could be designed, that would act as an intermediate HybridUML model representation. Therefore, the tool-specific model export and the conversion to a suitable HybridUML model for the transformation  $\Phi$  would be separated. Additionally, (experienced) users could create HybridUML models textually by use of a conventional text editor, and would not depend on the availability of a UML tool. A textual notation could be derived from the *Human-Usable Textual Notation (HUTN)* [OMGa] that is proposed for general UML. (3) The canonical way to separate model export from a tool and the succeeding conversion would be the customization and use of UML's *XML Metadata Interchange (XMI)* format [OMGe], with the loss of direct usability by humans.

These three ways are depicted on the upper left of figure 1.1. For this thesis, we assume the first, i.e. that there is an implemented assignment of graphical HybridUML specifications to valid HybridUML models. The graphical notation of HybridUML that we expect is illustrated in section 1.3 by means of a train control systems case study defined by the DFG priority programme *Software Specification – Integration of Software Specification Techniques for Applications in Engineering* [DFG]. It conforms to the profile definition of [BBHP03].

**HybridUML Expression Language.** A HybridUML specification consists, like general UML specifications, of graphical as well as textual elements. The syntax of the textual elements is given separately by the HybridUML Expression Language. Strictly speaking, the textual elements are part of the HybridUML syntax itself, but conceptually, different expression languages can be applied for UML as well as for HybridUML. In order to emphasize this, we separate the definition of the HybridUML Expression Language from the HybridUML syntax. In figure 1.1, this is depicted on the upper right. The HybridUML Expression Language is motivated and defined in chapter 3.

**Hybrid Low-Level Framework HL<sup>3</sup>.** The motivation of a low-level framework for the formal definition and execution of hybrid systems is two-fold: (1) The transformation definition of HybridUML models into arbitrary program code would only help to a limited extent – indeed, the resulting program would be defined unambiguously, but the feasibility problem of capturing the resulting behavior would remain. Our suggestion to overcome this is to *restrict* the infinite variety of possible compilation targets for hybrid specifications by use of the HL<sup>3</sup> framework. Therefore the behavioral semantics of the executable target can be given more easily than for an unrestricted compilation into a programming language. (2) We do not think that HybridUML is the only specification formalism ever that is suitable to model hybrid systems, and that fits to our transformation approach. Therefore, the HL<sup>3</sup> framework is designed such that it provides the building blocks we expect to be mandatory for hybrid systems modeling. Then, models of any adequate hybrid systems formalism can be transformed into HL<sup>3</sup> models, by a customized transformation  $\Phi$ .

The HL<sup>3</sup> framework consists of a pre-defined *design pattern* that restricts the software architecture of HL<sup>3</sup> programs, and a fixed re-usable hard real-time *runtime environment* which avoids uncertainties introduced by using arbitrary operating systems. A transformation  $\Phi$  then refines and instantiates the design pattern for the definition of the resulting HL<sup>3</sup> program, which relies on the runtime environment. Note that HL<sup>3</sup> is not meant to be used for manual programming, but as a target language for automated transformations. The HL<sup>3</sup> framework is defined in chapter 4.

**Transformation  $\Phi$  to HL<sup>3</sup> Programs.** The transformation  $\Phi$  is the algorithm that formally defines how a HybridUML model is mapped to a corresponding HL<sup>3</sup> program. It is discussed in chapter 5 in detail.

The transformation defined in this thesis maps HybridUML specifications to HL<sup>3</sup> *simulation* models. That means, the resulting HL<sup>3</sup> model represents the complete hybrid system as specified by the HybridUML model, including hybrid computer systems as well as their operational environment.

In order distinguish between the computer system and its operational environment, an *Architectural Specification* is needed. The HL<sup>3</sup> framework has a notion of *Interface Modules* to support this, such that parts of a HybridUML specification could be declared as “external” and therefore be excluded from code generation, but mapped to corresponding interface modules. An interface module is then an abstraction of some external (hardware) system, which must be defined manually, tailored to the specific system. For HybridUML, no architectural specification facility is defined, therefore the above distinction is not

supported (yet).

Further, the HL<sup>3</sup> framework supports an explicit assignment of required frequencies for the discretized calculations of continuous behavioral parts. This would be defined by use of a *Physical Constraints Specification*, attached to the HybridUML model. So far, there is no specification facility for this, either, such that the transformation  $\Phi$  will not distinguish different frequencies.

**Contents of this Thesis.** This thesis focuses on the transformation concept and the semantic model of the HL<sup>3</sup> framework, rather than on the technical details of hardware/software distribution over processors and clusters, interfacing of different (sub-)systems, or tuning of the executable system.

*Prerequisites.* The transformation *concept* from hybrid systems specification formalisms to executable models of a low-level language has been joint work of the authors from [BBHP04a]. There, the concept has been proposed informally, and the HL<sup>3</sup> Low-Level Framework has been sketched.

The same authors have given a semi-formal definition of HybridUML in [BBHP03]. There, the graphical syntax of HybridUML has been presented, and some informal semantics have been given. The specific transformation from HybridUML to HL<sup>3</sup> has been illustrated in [BBHP04a].

*Contributions.* Based on these prerequisites, this thesis contains three main contributions: (1) Based on the HybridUML graphical syntax of [BBHP03], a mathematically formal HybridUML syntax is defined. It is complemented by the formal definition of the HybridUML Expression Language. (2) Based on the initially informal proposal in [BBHP04a], a full definition of the HL<sup>3</sup> Low-Level Framework is constructed. This contains a complete formal operational semantics for it. (3) The specific transformation  $\Phi_{HUML}$  is defined formally. This maps HybridUML models to executable HL<sup>3</sup> models and therefore defines an operational semantics for HybridUML models, along with an executable model simulation.

For the proof of concept, we have implemented the HybridUML Mathematical Meta-Model, the HybridUML Expression Language, the transformation  $\Phi_{HUML}$  itself, and a subset of the HL<sup>3</sup> Low-Level Framework. All implementations are coded with C++ (or generated from the scanner/parser generator tools `flex` and `bison`). The implemented transformation results in a C++ variant of the defined HL<sup>3</sup> program that can be compiled and executed.

As an application example, a realistic case study from the field of train control systems is applied, which has been derived from a respective requirements specification of the Deutsche Bahn AG (German Railways). The complete behavior of the train, as well as the complete behavior of railroad crossings is defined. A graphical model representation is given in appendix C, and an instance of the implemented HybridUML Mathematical Meta-Model is used (but not presented) to test the implemented transformation  $\Phi_{HUML}$ . Excerpts of the C++ variant of the resulting HL<sup>3</sup> program are shown in appendix D.

**Outline.** This introductory chapter further provides references to related work in section 1.2, and closes with an intuitive presentation of graphical HybridUML

specifications in section 1.3. For this, the train control case study is applied. The complete HybridUML model of the case study is contained in appendix C.

In chapter 2, the syntax of HybridUML specifications, i.e. the HybridUML meta-model, is defined. Chapter 3 provides the allowed textual specification parts that are embedded into HybridUML specifications by means of the HybridUML Expression Language. The Hybrid Low-Level Framework is discussed in chapter 4, followed by the definition of the transformation  $\Phi$  for HybridUML models in chapter 5. A conclusion is given in chapter 6.

Appendix D contains excerpts of the C++ variant of a HL<sup>3</sup> program which is generated by the implementation of the transformation  $\Phi$  – the transformation result of the case study’s HybridUML model of appendix C.

## 1.2 Related Work

In this section, we give a short summary of related work to our transformational approach. Of course, investigations on hybrid systems and real-time systems in general affect our work, as well as UML and similar graphical specification languages. Additionally, semantics of programs are fundamental to our operational semantics of HL<sup>3</sup> programs.

Finally, references to the specification of train-control systems are given, that correspond to the case study presented in section 1.3.

**HybridUML and HL<sup>3</sup>.** HybridUML was initially defined in [BBHP03] as a UML 2.0 profile. The semantics given there were not based on our transformation concept, but were defined rather informally. In [BBHP04a], the transformation approach along with the corresponding low-level framework HL<sup>3</sup> were proposed. Summaries were presented in [BBHP04b] (in German) and [BZL04].

**Hybrid Systems.** The main source of inspiration for HybridUML is the specification formalism *CHARON* [AGLS01, ADE<sup>+</sup>01, ADE<sup>+</sup>03], which extends the concept of *Hybrid Automata* [ACHH93, NOSY93, ACH<sup>+</sup>95, Hen96]. While Hybrid Automata demonstrated the feasibility of model checking for hybrid specifications, CHARON improved the applicability of hybrid automata to large-scale systems by introducing hierarchy for hybrid specifications.

Alternative hierarchical approaches closer to the Statecharts formalism have been described in [BBB<sup>+</sup>99] and [KP92, KMP00]. The latter distinguish between timed and hybrid systems, where timed systems are special hybrid systems that only allow clocks to evolve continuously. *Timed Automata* were introduced in [AD94].

In [LSV03], *Hybrid I/O Automata* are discussed, which explicitly distinguish between input and output actions as well as variables.

Modeling of hybrid systems based on petri nets is discussed by means of *Extended Coloured Petri Nets* in [YLB94].

The *Duration Calculus* is a noteworthy formalism in that it contributes fundamentally to the understanding of hybrid systems [ZRH93, RRS03].

Further investigations in the field of hybrid systems have been done in the context of the DFG priority programme Kondisk [EFS02]. The formalism *UML<sup>H</sup>* discussed in [Nor03] (in German) is based on UML, like HybridUML.

Its semantics is defined by use of Object-Z [RGG94], an extension to the formal language Z [Spi92].

The visual formalism *HyCharts* for the description of hybrid systems is introduced in [GS98].

On the basis of the process algebra *CSP* [Hoa85], the extension *HybridCSP* is discussed in [Jif94, CJR95]. An operational semantics is given in [Amt99]: HybridCSP specifications get their semantics by a decomposition into a *Timed-CSP* part and a set of so-called HybridCSP Patterns. In the same fashion, TimedCSP is defined by a decomposition into an untimed CSP part and a set of TimedCSP patterns, as discussed in [Mey01].

**Tools-Based Approaches.** The tool *Mathlab* [MAT] is widely used to perform numeric computation for diverse mathematical problems. It is complemented by the *Simulink* [Sim] component, which provides graphical means to simulate and implement real-time systems.

The language *Modelica* [EMO] supports development and simulation of physical systems, supported by the non-profit Modelica Association. Besides several commercial tools, the open source project *OpenModelica* [Ope] aims at a complete modeling, compilation and simulation environment.

**Statecharts.** Hierarchical state machines for discrete systems were introduced as *Statecharts* in [Har87]. There are a lot of statecharts variants, a (slightly outdated) survey was given by [vdB94]. The tool *STATEMATE* [HN96] provides simulation as well as code generation capabilities. A formal semantics is given in [DJHP98].

Statecharts also provide the basis for *UML State Machines*, which are the main concept of UML for the specification of behavior. They are included in the UML specification documents cited below.

**UML.** The *Unified Modeling Language (UML)* is defined by textual specification documents [UML, OMGc], provided by the Object Management Group. The “definitive description of UML from its original developers” is provided by [RJB04]. For the definition of UML, a subset of UML itself – the *Meta-Object Facility* [OMGb] – is given separately.

The benefit of UML to the analysis and design of real-time systems is discussed in [Dou99, Dou04]. In [Sel98], a set of structural modeling concepts for real-time systems are proposed as extensions to UML (version 1.x).

Based on version 2.0, UML comes with the *UML Profile for Schedulability, Performance, and Time* [OMGd] which attempts to define standard paradigms of use for modeling of time-, schedulability-, and performance-related aspects of real-time systems.

Finally, a critical review on the UML support of real-time systems is given in [Ber03].

**Real-Time Systems.** For the analysis, design, development, and verification of real-time systems, numerous contributions exist in the literature. We would like to point out [Kop97] as our main reference on real-time systems. Particularly, the *time-triggered approach* for the design of embedded real-time systems is motivated, which is well-suited for real-time programs discretizing

time-continuous evolutions, since it guarantees bounded timing jitter for periodic schedules.

The fundamentals of safety-critical systems, which are a major subset of today's hard real-time systems, are comprehensively presented in [Sto96]. Contributions to the analysis, design and verification, as well as to program structures of real-time systems are contained in [Rav95] or [Jos95], for example, along with further references.

**Real-Time Systems Programming.** For the development of discrete systems, real-time programming languages exist. *GIOTTO* [HHK03] is today's most prominent example of a hard real-time language with well-defined semantics, which follows the time-triggered systems paradigm described in [Kop97].

Further, languages like *Esterel* [BG92, Ber00], *Lustre* [HCRP91], or *Signal* [GGBL91] facilitate synchronous programming of real-time systems. They are related to Statecharts in that they share the notion of synchronous concurrency [Hal92]. Specific Statechart variants are *Argos* [MR01] and *SyncCharts* [And96], an extension of Argos. For SyncCharts, a translation to Esterel exists [And96], but seemingly is not defined formally. Esterel, Lustre, and a variant of SyncCharts are integrated into the commercial toolkit *SCADE Suite* [Sca], which is used for the development of safety-critical software in the avionics domain, for example.

With respect to the level of abstraction, the above mentioned real-time programming languages, including GIOTTO, relate to our HL<sup>3</sup> framework, whereas the role of SyncCharts is more alike HybridUML for our hybrid systems approach.

**Real-Time Systems Implementation.** The execution of real-time systems of course requires adequate hardware, and potentially an appropriate real-time operating system. In [Ott06], a comparison of commercial-off-the-shelf (COTS) and special-to-purpose platforms is given. Examples for COTS hardware are PCs (with or without standardized components), or programmable logic controllers (PLC). Among COTS operating systems, there are VxWorks [VxW] or RTLinux [RTL]. A popular non-commercial variant of the Linux operating system is RTAI [RTA].

We will not discuss the *implementation* of our HL<sup>3</sup> framework and the corresponding HL<sup>3</sup> programs created from the transformation  $\Phi$  in detail, a brief overview is postponed to chapter 6. Note here, that is based on standard Linux, with a real-time scheduler modification [Efk05] that provides CPU reservation as well as periodic scheduling and therefore supports the time-triggered systems paradigm of [Kop97].

**Semantics of Programs.** A formal semantics for programs is defined in [AO97]. For the definition of the operational semantics of HL<sup>3</sup> programs, we *will rely* on the definitions given there. We will summarize them only briefly in section 4.6. For the creation of programs, the guidelines given in [Dij76] are still valid.

**Train Control.** The HybridUML model presented in section 1.3 is based on the case study "Radio-Based Train Control" [JS00, HPSS04] provided in the

context of the DFG priority programme Software Specification [DFG]. The case study description itself is derived from a respective requirements specification [FFB96] of the Deutsche Bahn AG (German Railways).

Background information on railway operation, signaling principles and traffic control technologies are given in [Pac02].

A major aspect of the case study is to guarantee that trains only pass railroad crossings when they are *safe*, which is a well-studied problem in the literature, for example discussed in [HL94].

In [BZL04], we applied a related case study to HybridUML – the BART case study [WB01]. BART provides a commuter rail service for part of the bay area of San Francisco, California.

## 1.3 HybridUML by Example: Radio-Based Train Control

This section introduces HybridUML by means of an application example – the specification of a radio-based train control system [JS00, HPSS04]. This is one of two case studies within the scope of the DFG priority programme Software Specification [DFG]. It is based on a real-world requirements specification that aims to the coordination of trains and railroad crossings, including the train control computer system on the one hand, and the railroad crossing controller on the other. The operational environment consists of a train moving on a track, and railroad crossings located on the track. Both consist of a number of components which describe the significant physical aspects that affect the controllers, or are affected, respectively.

The resulting train controller and crossing controller models are not just small academic examples. Unlike the conventional thermostat or leaky water tank examples, the given design covers a significant part of the functionality that would be utilized in real train and crossing controllers. For the execution of the resulting HL<sup>3</sup> model, quite powerful hardware (like state-of-the-art PC hardware) is adequate.

In this section, the concrete application-specific design is shown in order to illustrate how HybridUML is used to (1) create the static structure as well as to (2) define the behavior of the model. The complete HybridUML model is given in appendix C.2, containing the full definitions of the controllers, as well as of their operational environment. Parts of the resulting HL<sup>3</sup> model are presented in appendix D.

### 1.3.1 Radio-Based Train Control

The case study deals with a decentralized radio-based control system that consists of a train, railway level crossings, and an operations center. The most important part of the specification of the control system is the coordination of train and railroad crossings in order to ensure that whenever the train crosses the road, the crossing is safe (i.e. it is locked for cars, pedestrians etc.). In addition to related investigations concerning the “generalized railroad crossing” (see [HL94], for example), the special feature of the case study is the absence

of signals and train monitoring equipment on the track. Train and crossings always have to guarantee a safe and consistent state of the complete system on behalf of state requests and notifications, which are communicated by means of mobile radio communication. Particularly, the train controller continuously (re-)calculates velocity-dependent locations on the track at which requests must be sent, or at which the brake has to be activated.

In this case study, a single train is considered that moves on a single track without switches. There are *velocity target points* on the track with dedicated velocities that the train must not exceed. There are two kinds of velocity target points: (1) *Conditional* velocity target points are assigned to railroad crossings and have a target velocity  $v = 0$ . If the crossing is not safe, the velocity target point is active and the train has to stop there. (2) The route atlas contains a (piecewise constant) static velocity profile that defines a maximum allowed velocity for each location on the track. Each location for which this velocity gets lower implies a *fixed* velocity target point, denoting a restrictive velocity change. These points are always active.

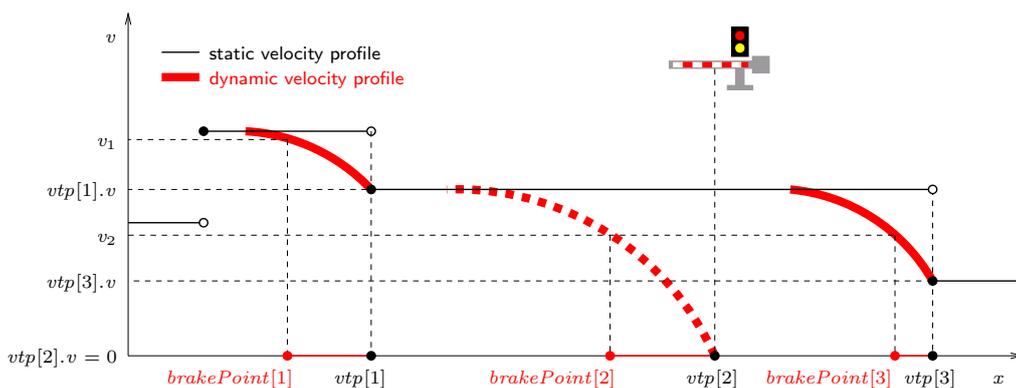


Figure 1.2: Dynamic velocity profile defined by velocity target points on the track

Figure 1.2 is an abstract view on a track example with fixed ( $vtp[1], vtp[3]$ ) and conditional ( $vtp[2]$ ) velocity target points. In consideration of the train's maximum deceleration, the static velocity profile and the velocity target points define the *dynamic* velocity profile, resulting in *brake points* – the locations on the track where the train must start braking in order to reach the target velocity at the respective target point. The brake points depend on the train's current velocity; in the diagram, example values for current velocities  $v_1$  and  $v_2$  are given, whereas  $v_1$  leads to brake point  $brakePoint[1]$  and  $v_2$  implies  $brakePoint[2]$  and  $brakePoint[3]$ , respectively.

The major safety-critical aspect of the train control system is to ensure that the speed limit of target velocity points is never broken. This includes that whenever the train crosses the road, the crossing is safe. The remainder of this section focuses thereon. For a more detailed description of the case study, see appendix C.1.

### 1.3.2 HybridUML model: Radio-Based Train Control

The graphical notation of HybridUML provides three types of diagrams, in order to specify the *static structure*, as well as the *behavior* of a hybrid system:

*Class Diagrams* contain separate definitions of the main structural building blocks of a HybridUML specification – so-called *agents*, which are specialized UML classes.

*Composite Structure Diagrams* define the internal structure of agents, which may contain instances of other agents.

*Statechart Diagrams* are used to define the behavior of agents. For each agent that has no internal structure, the behavior is given by exactly one state-machine.

#### Structural Specification

**Agent.** The main building block for modeling static structure within HybridUML is called *agent*. Agents are specialized UML classes, and therefore define a set of objects of similar type. Every agent defines a set of variables (also called properties), either accessible only locally, or also publicly. The latter can be used to model shared variables for a set of objects. Further, a set of signal specifications is given for agents, which provide an instantaneous message passing mechanism. Variables and signals constitute the (precise) interface description of agents.

The objects that are defined by agents are concurrently operating entities: *Basic agents* are agents that have sequential behavior. In contrast, *composite agents* have *internal structure*, defined by objects of other agents. The complete system is defined as a composite agent *System*: In Fig. 1.3, the class diagram view is shown, which defines an empty set of variables and an empty set of signals.



Figure 1.3: Agent System.

The internal structure is defined in Fig. 1.4. There, the train, the operations center, and a set of crossings are contained, by means of *agent instances*. The radio channels for the communication between them are modeled explicitly, too.

Particularly, the train is part of the complete system, therefore the agent *Train* defines its structure: (1) Signal specifications are given for the receiving and sending of telegrams, for the communication with the operations center and the crossings. (2) The train's position  $x$  on the track is defined as a public variable. Both are defined in Fig. 1.5. (3) The internal structure of the train is given as composite structure diagram in Fig. 1.6. Its components are the user (locomotive driver), the train's localization device, the brake, the engine; the train's movement is modeled explicitly and internally encodes the relation between the accelerations and the resulting velocity. Finally, the train controller is part of the train.

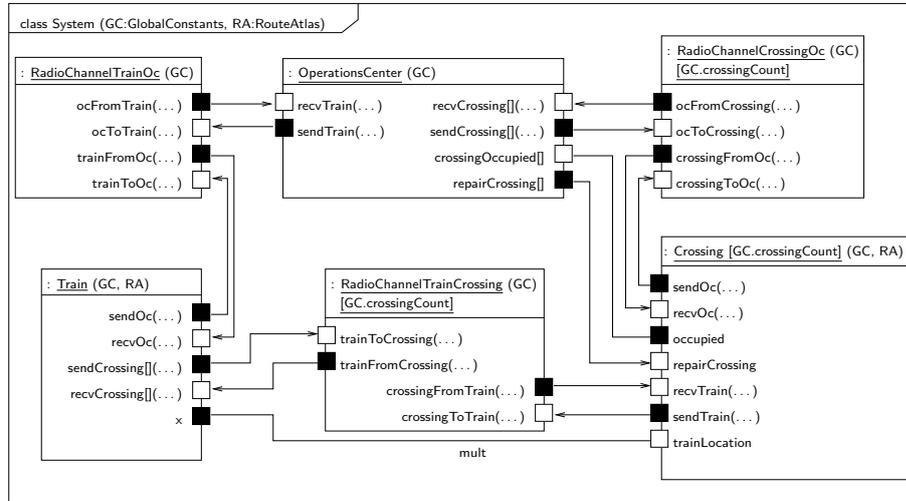


Figure 1.4: Structure diagram of agent System.

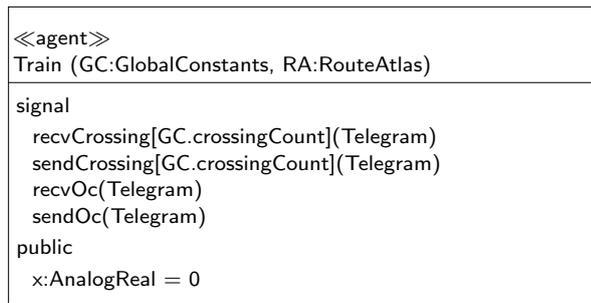


Figure 1.5: Agent Train.

Note that the environmental specification is modeled in the same fashion as the computer system. Therefore, both discrete and continuous specification elements are available (see the behavioral specification below) for the modeling of the environment as it is, rather than a discretized interpretation thereof. Further, the computer system observes and affects this environment, which ideally would take place infinitely fast and thus would proceed approximately time-continuously. The deviation of a real computer system's discrete implementation from the ideal time-continuous execution is mainly given by the its hardware capabilities, therefore it is desirable to create a model that abstracts from the particular hardware restrictions, in the same way as the environment is modeled.

Agent TrainController consists of several agent instances as defined in its structure diagram (Fig. 1.7), including BrakePointController and MovementController (Fig. 1.11 and Fig. 1.12). TrainController calculates and provides the required acceleration  $a$  of the train. Therefore the train's location  $x$  and the user-requested acceleration  $a_{\text{user}}$  from the locomotive driver are read from the environment. These are time-continuously changing variables. Further, radio messages are received that provide status information about the railroad crossings.

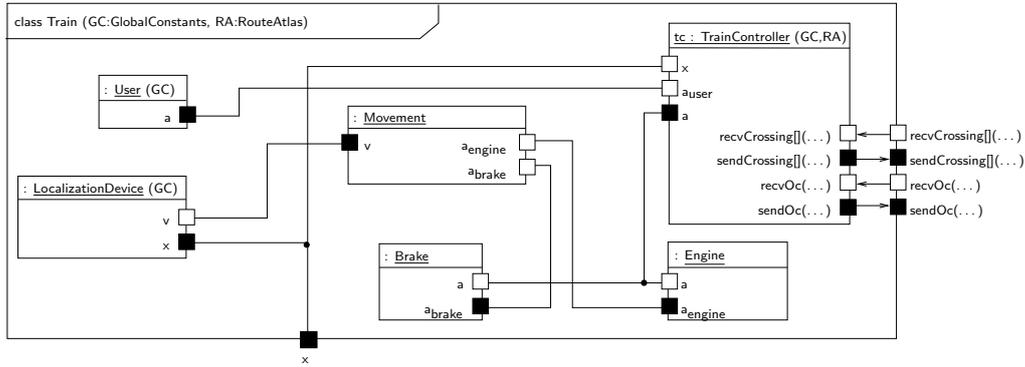


Figure 1.6: Structure diagram of agent Train.

The variables and signals are defined in the class view of Fig. 1.8. It contains locally defined constants which affect the computation of  $a$ , like the route atlas  $RA$  which is defined as a data structure. It contains the velocity target points  $vtp$  that consist of a location  $x$ , a target velocity  $v$ , and its type (fixed or conditional). The definition of data structures is done by special classes, shown in Fig. 1.9. For `TrainController`, they are used as *parameters*, which are special local variables that are constant, i.e. constants, and which have a special parameter notation.

**Agent Instance.** Agent instances define instances of a corresponding agent, i.e. they define objects of a respective class. From the definition of the complete system `System`, a single instance represents the particular system, defined in Fig. 1.10.

Apart from this special object, agent instance definitions occur within composite structure diagrams, in order to define the internal structure of agents. Figure 1.4 defines the internal structure for *all instances* of `System`, since it is part of the definition of the *agent*. Each instance of `System` then contains one instance of `Train` and `OperationsCenter`. Note that a single agent instance specification therefore implicitly defines a *set of instances*. Further, for a single instance of `System`, the agent instance specification of `Crossing` (Fig. 1.4) defines more than one object: within brackets, a multiplicity is defined which denotes the actual number `GC.crossingCount` of crossings, resulting in `GC.crossingCount` crossing objects.

For the parameters of the agents `Train`, `OperationsCenter`, `Crossing` (, and the radio channels as well), expressions are given that act as actual parameters. In the example,  $GC$  and  $RA$  are themselves (formal) parameters in the context of *System*, and thus can be used within the actual parameter expressions. In the same way, the above multiplicity of `Crossing` technically is an expression. The applied expressions must be evaluable on construction of the system, therefore they are restricted to contain literals and parameters.

**Property.** Variables of agents (which we synonymously call *property* for compatibility with UML) have a multiplicity that is defined by an expression (like for agent instances). We require a fixed multiplicity, therefore a property can be interpreted as an array of values.

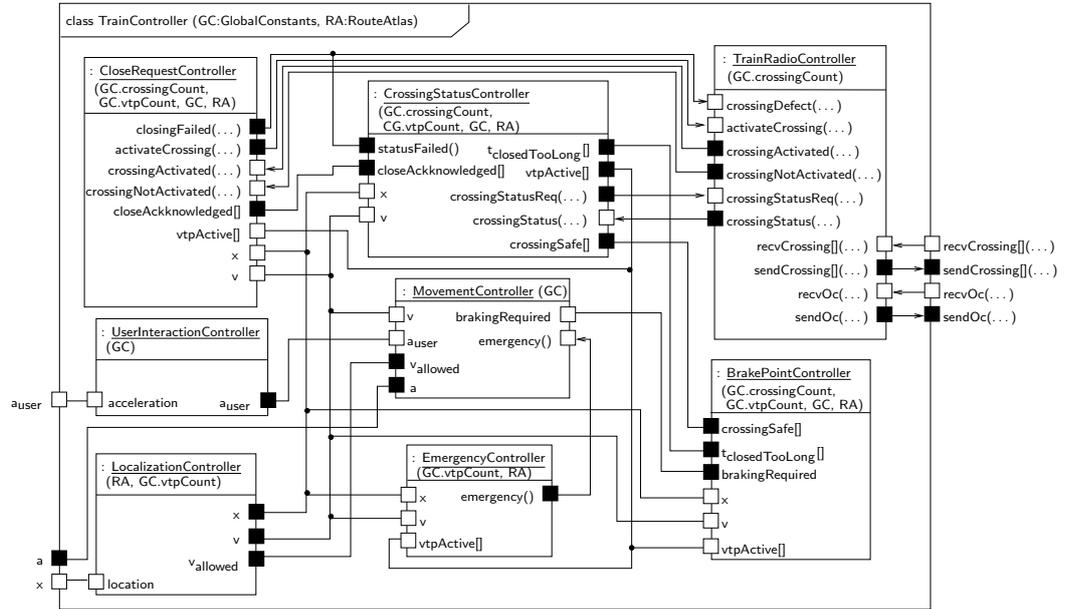


Figure 1.7: Structure diagram of agent TrainController.

For example, the local property `brakePoint` of agent `BrakePointController` (Fig. 1.11) has the multiplicity `VTP_COUNT`, such that for each velocity target point, the corresponding brake point is stored. Variables without an explicit multiplicity have always the multiplicity 1.

The visibility of a variable (`private`, i.e. local, or `public`) defines if the variable is only accessible locally by the containing object, or if it may be shared with other objects. This is discussed below, for variable ports.

Finally, a variable has an access policy, which defines whether it can be modified or is constant. The latter is denoted by `{read-only}`.

**DataType.** HybridUML variables are typed. Primitive types are predefined, there are `Boolean`, `Integer`, `Real`, and `AnalogReal`. Variables of type `AnalogReal`

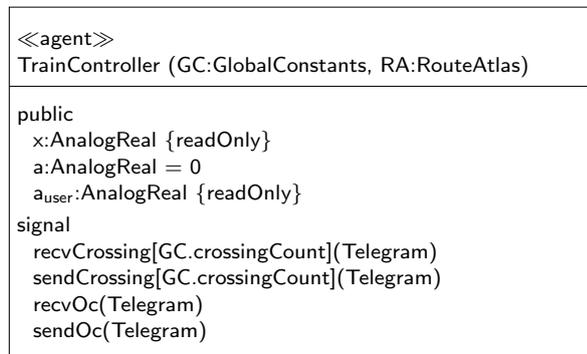


Figure 1.8: Agent TrainController.

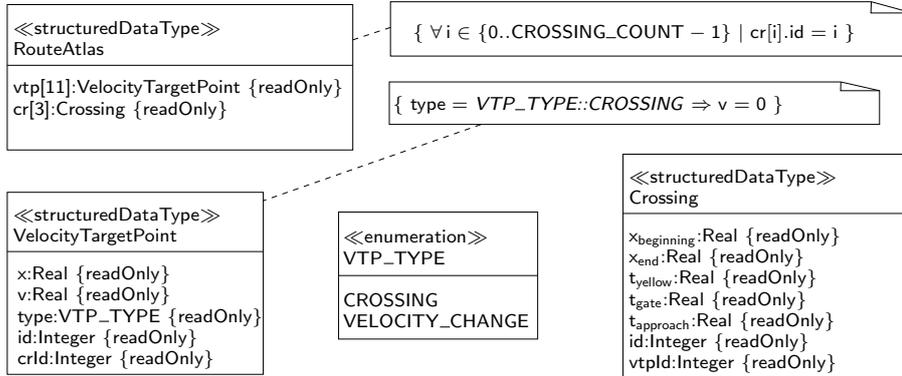


Figure 1.9: Structured and Enumeration Data Types defining the Route Atlas.



Figure 1.10: Agent Instance System.

can be modified continuously, like `brakePoint` above.

Additionally, user-defined data types can be defined by means of (1) structured data types, and (2) enumeration data types. The route atlas, which is used above as agent parameter, is defined as a structured data type in Fig. 1.9, which is a special class without behavior, containing only public variables. The route atlas contains information about crossings and velocity target points, each themselves modeled as structured data types. Velocity target points have a `VTP_TYPE`, which denotes whether it is associated with a crossing, or with a restrictive velocity change. This type is defined as enumeration type, that is also a special class, containing only constants that act as enumeration literals.

**PropertyValue.** Variables optionally have an initial value, which is defined by a property value specification, given as an expression. For example, the variable `brakePoint` of agent `BrakePointController` (Fig. 1.11) is initialized with 0. This affects all indices wrt. to the multiplicity `VTP_COUNT`. For parameters, the initial value (i.e. the actual parameter) is mandatory, since it defines the parameter's constant value. The route atlas is explicitly given for top-level agent

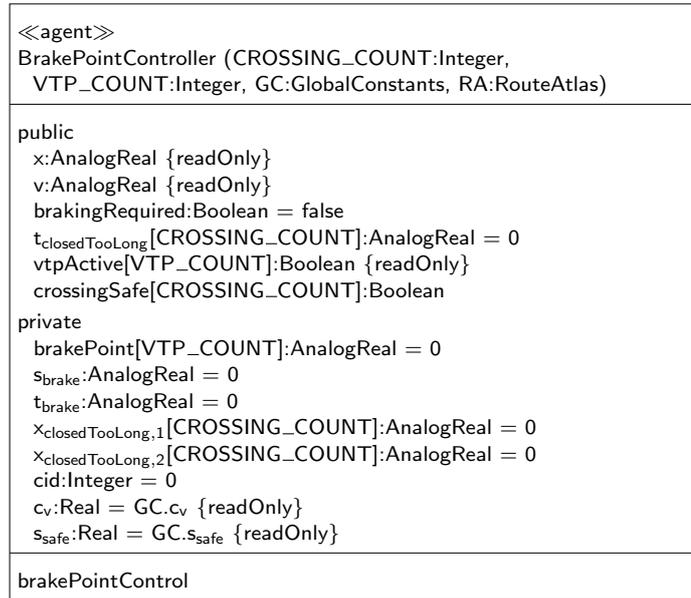


Figure 1.11: Agent BrakePointController.

instance of System (Fig. 1.10):

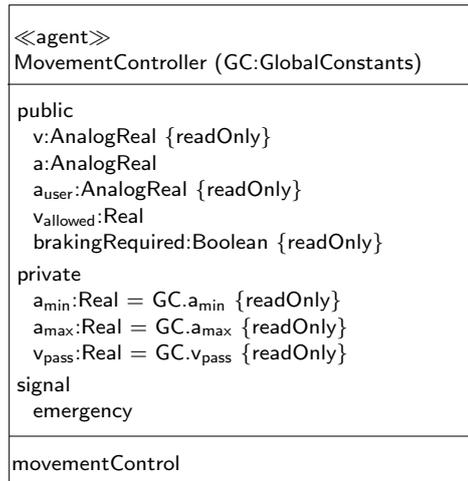
```

(C.2) actprmRA ≡
{
  {0, 27.8, VTP_TYPE::VELOCITY_CHANGE, 0, 0},
  {1600, 16.7, VTP_TYPE::VELOCITY_CHANGE, 1, 0},
  {3000, 33.3, VTP_TYPE::VELOCITY_CHANGE, 2, 0},
  {4200, 30.5, VTP_TYPE::VELOCITY_CHANGE, 3, 0},
  {5200, 11.1, VTP_TYPE::VELOCITY_CHANGE, 4, 0},
  {5900, 9.7, VTP_TYPE::VELOCITY_CHANGE, 5, 0},
  {6400, 22.2, VTP_TYPE::VELOCITY_CHANGE, 6, 0},
  {7100, 16.7, VTP_TYPE::VELOCITY_CHANGE, 7, 0},
  {2800, 0, VTP_TYPE::CROSSING, 8, 0},
  {4960, 0, VTP_TYPE::CROSSING, 9, 1},
  {7205, 0, VTP_TYPE::CROSSING, 10, 2},
  {2800, 2840, 6, 12, 13, 0, 8},
  {4960, 5000, 6, 12, 13, 1, 9},
  {7205, 7245, 6, 12, 13, 2, 10}
}

```

This defines a route with 11 velocity target points and 3 crossings. Each velocity target point has an absolute position on the track, an associated maximum velocity, a type, an identifier, and optionally an associated crossing identifier. The crossings are defined by a beginning and an end position, some timing parameters, an identifier, and the identifier of the corresponding velocity target point.

**VariablePort.** As stated before, public variables of agents (more exactly, of the agent's objects) can be shared between objects. This is defined by *variable connectors* that connect *variable ports*, which represent public variables. In the lower right of Fig. 1.7, the agent instance specification of `BrakePointController` defines a read-write port for the variable `brakingRequired`, which itself is defined in Fig. 1.11. A port specification can restrict the access policy to read-only: the agent instance of `MovementController` (in the middle of Fig. 1.7) defines read-only access for its local variable of the same name.

Figure 1.12: Agent `MovementController`.

**VariablePortInstance.** The explanations above are somewhat inexact, because there is a subtle difference between *variable ports* and *variable port instances*. A variable port is defined for the agent, and therefore for all objects of the agent, whereas a variable port instance is only associated with the subset of objects that are specified by the agent instance. The difference between variable ports and variable port instances is important for the interpretation of connectors, as discussed below.

In Fig. 1.7, variable port instances are defined for `brakingRequired` of `BrakePointController` and `MovementController`. They *imply* that a corresponding variable port exists, too, because each variable port instance represents a variable port for the given agent instance specification.

Note that the specifications for variables `a` and `x`, for example, on the lower left of Fig. 1.7 define variable ports only, because they belong to the agent definition of `TrainController`, rather than to an agent instance specification thereof.

**VariableConnector.** A variable connector connects a set of variable port instances and up to one variable port. Therefore it defines a set of variables from a set of objects that shall act as a *single shared variable*. The variable port instances of the variables `brakingRequired` of `BrakePointController` and `MovementController` in Fig. 1.7 are connected by a variable connector, and therefore model a shared variable. `BrakePointController` objects can read and write it, whereas `MovementController` objects only have read access.

More than two port instances can be connected, for example the observed velocity  $v$  which is provided by `LocalizationController` is read by almost all components of `TrainController`.

Variables of composite agents can be shared with contained objects, therefore a connector may be attached to a variable port as well. For example, the controlled acceleration  $a$  of `TrainController` is the acceleration that is calculated by its agent instance `MovementController`.

Variable connectors additionally have a connector type. Normally, a connector is a *point-to-point* connector, with respect to the multiplicities of the connected variables and their containing agent instances (via port instances and ports). For example, the `CrossingStatusController` (Fig. 1.7) provides a variable `vtpActive` with a multiplicity of `VTP_COUNT`, such that for each velocity target point, a boolean value denotes whether it is currently active or not. A velocity target point may be inactive, when a corresponding crossing is currently safe, i.e. the gates are closed. A variable connector defines that `BrakePointController` can read these values (Fig. 1.7), but of course, each index of `vtpActive` shall be mapped separately, resulting in `VTP_COUNT` shared variables, rather than in a single one.

In contrast, there are situations for which all indices of variables shall be merged into a single shared variable. For example, in Fig. 1.4, all crossings have access to the train's location  $x$ . Therefore,  $x$  is connected to `trainLocation` of the crossings by a *multicast* connector.

**Signal.** Signal specifications have, like variable specifications, a fixed multiplicity that is defined by an expression, therefore a signal can be interpreted as an array of signal values. A signal does not have a visibility or an access policy: There are no local signals, since they are always used for communication *between* objects, and the access policy is defined by signal ports (see below).

Instead of a single data type, a signal has a *list of data types*<sup>3</sup> which corresponds to the signal parameter values that the signal can carry. For example, in Fig. 1.8 the `TrainController` defines signals that it can handle, each carrying a Telegram value.

**SignalPort.** The sharing of signals is defined in the same way as for variables: *Signal connectors* connect *signal ports*, which represent signal specifications of objects.

The signal *emergency* is shared between `EmergencyController` and `MovementController` (Fig. 1.7). Signals are either write-only (i.e. they are *sent*), or read-only (i.e. they are *received*). The signal port of `EmergencyController` is a send port, whereas the signal port of `MovementController` is a receive port. Therefore, the direction of the shared signal is defined.

**SignalPortInstance.** As for variable ports and variable port instances, exactly the same distinction between *signal ports* and *signal port instances* exists. Therefore, we do not repeat it here.

---

<sup>3</sup>This could be denoted as the signal's *signature*.

**SignalConnector.** Signal connectors work the same way as variable connectors do, i.e. they connect a set of signal port instances and up to one signal port. The distinction between point-to-point connectors and multicast connectors exists, too.

In Fig. 1.7, a signal connector actually defines that the signal *emergency* is shared between **EmergencyController** and **MovementController**. The arrowhead only emphasizes the direction of the signal, the definition is given by the access policies of the port instances.

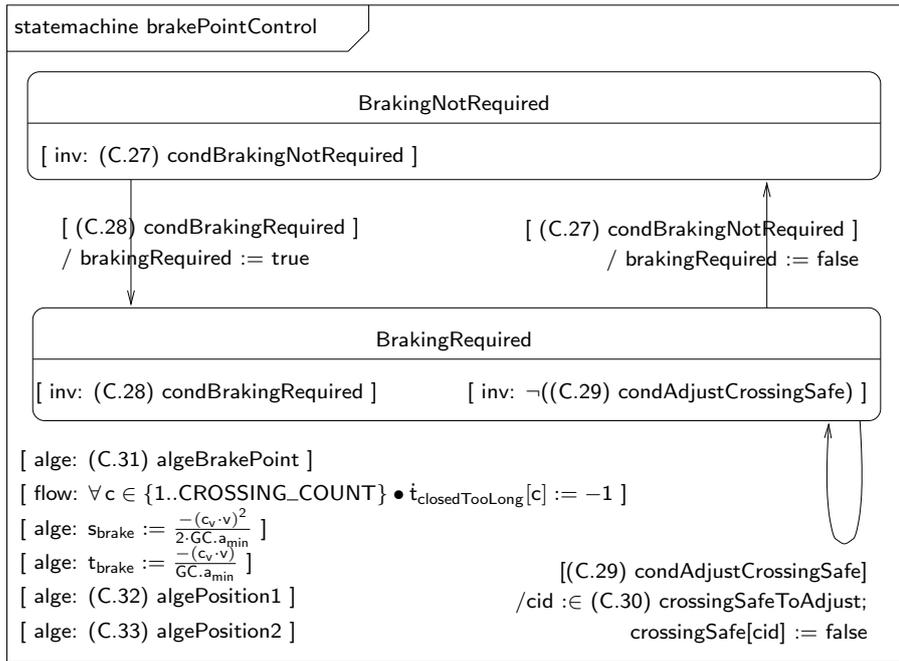


Figure 1.13: Behavior of agent BrakePointController.

### Behavioral Specification

Embedded into the structural specification parts, the behavioral specification is defined by means of state-machines. Each basic agent has an attached sequential state-machine, therefore basic agents define active objects in terms of UML. In contrast, a composite agent has no own state-machine, but it always contains – directly or indirectly – a set of basic agents. Therefore, the behavior of a composite agent is the concurrent execution of the sequential behavior of all contained basic agents.

**Mode.** State-machines are defined by means of *modes*. For example, the behavior of agent **BrakePointController** (see Fig. 1.11) is defined by the mode **brakePointControl**, shown in Fig. 1.13. The main responsibility of **BrakePointController** is to provide the boolean variable **brakingRequired** that denotes that the train has to brake because of any velocity target point.

In order to determine `brakingRequired`, the train's location `x`, its current speed `v`, and the set of currently active velocity target points (which is defined by `vtpActive[]` consisting of a boolean variable for each velocity target point), are read.

As typical for hybrid systems, there are basically two ways of acting: either some discrete transition is taken, conceptually performed in zero-time, or time passes and the continuous variables change over time according to their specified constraints. The behavior definition (shown in Fig. 1.13) is based on the continuous (re-)calculation of the set of brake points `brakePoint[]` for all velocity target points:

$$(C.31) \quad \mathbf{algeBrakePoint} \equiv \\ \forall i \in \{0..VTP\_COUNT - 1\} \bullet \\ \mathit{brakePoint}[i] := RA.vtp[i].x - \frac{RA.vtp[i].v^2 - (c_v \cdot v)^2}{2 \cdot GC.a_{min}} - s_{safe}$$

Continuous calculations are defined by algebraic and flow constraints, which are given as expressions labeled with `alge` and `flow`, respectively. Flow constraints may directly use the first derivative of a variable, e.g.  $\dot{t}_{closedTooLong}[c]$  in Fig. 1.13.

The variable `brakingRequired` is set in a discrete fashion, dependent on condition:

$$(C.28) \quad \mathbf{condBrakingRequired} \equiv \\ \exists i \in \{0..VTP\_COUNT - 1\} \bullet \\ (\mathit{brakePoint}[i] \leq x \wedge RA.vtp[i].v < v \wedge RA.vtp[i].x > x \wedge \\ (\mathit{vtpActive}[i] \vee \\ RA.vtp[i].type = VTP\_TYPE::CROSSING \wedge \\ ((x_{closedTooLong,1}[RA.vtp[i].crId] \\ \leq RA.cr[RA.vtp[i].crId].x_{end} + GC.trainLength \\ \wedge t_{closedTooLong}[RA.vtp[i].crId] \leq t_{brake}) \\ \vee (x_{closedTooLong,2}[RA.vtp[i].crId] \\ \leq RA.cr[RA.vtp[i].crId].x_{end} + GC.trainLength \\ \wedge t_{closedTooLong}[RA.vtp[i].crId] > t_{brake}))))))$$

The condition (C.28) `condBrakingRequired`, as well as its negation (C.27) `condBrakingNotRequired` are used (1) as guard conditions of transitions, as well as (2) as invariant constraints (labeled with `inv`) of modes. The transitions in combination with the invariants model the mandatory mode changes according to the condition, such that variable `brakingRequired` is always up-to-date. It denotes the situations that require braking because at least one brake point of an active velocity target point is reached by the train while its speed is too high. Note that only velocity target points in front of the train are considered, because particularly the opening of a crossing behind the train shall not affect it. Thus the mode `BrakingRequired` is active for exactly these situations, whereas the mode `BrakingNotRequired` is complementary.

**ModeInstance.** Modes are *hierarchical* state-machines, in that they contain *mode instances*. The distinction of modes and mode instances is that a mode *defines* a set of state-machines, and a mode instance actually *is* one of these state-machines, in a similar way as objects are instances of classes. The state-chart diagram of Fig. 1.13 defines a mode `brakePointControl` that contains two

submode instances `BrakingNotRequired` and `BrakingRequired`. The modes for `BrakingNotRequired` and `BrakingRequired` are defined in-place, which is suitable for modes that are instantiated only once. So far, the distinction between mode and mode instance is totally transparent.

In contrast, for the behavior definition of agent `MovementController` (Fig. 1.12), the mode definition of `movementControl` (Fig. 1.14) instantiates the submode `braking` twice, therefore the mode is defined separately (Fig. 1.15).

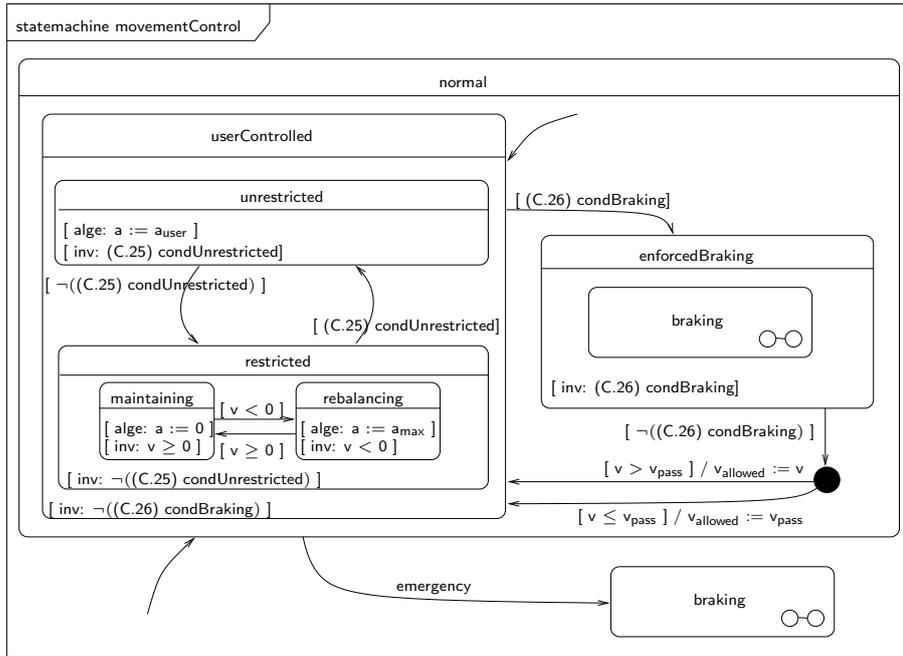


Figure 1.14: Behavior of agent `MovementController`.

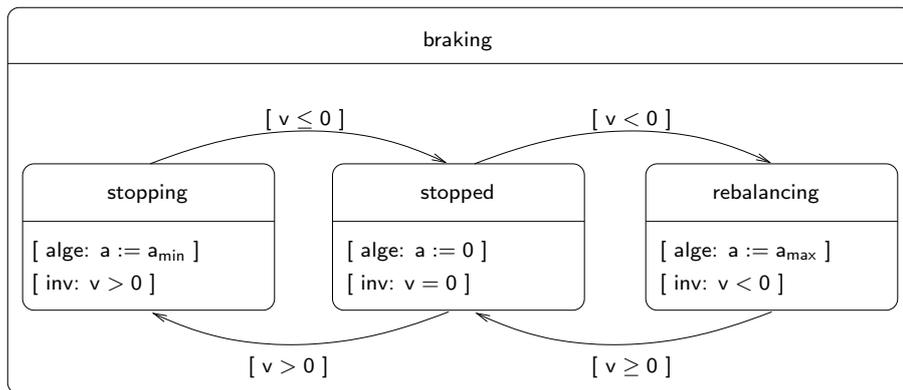


Figure 1.15: Definition of submode `braking`.

The responsibility of agent `MovementController` is to determine the required acceleration `a`. It constrains the user-requested acceleration `auser` on behalf of

`brakingRequired`, the current velocity  $v$ , the currently allowed velocity  $v_{\text{allowed}}$ , and a special signal `emergency`. It is modeled strictly hierarchically, initially in mode `normal`. Submode `userControlled` maps the user-requested acceleration to  $a$ , whereas submode `enforcedBraking` forces the train to brake. Similarly to `BrakingRequired`, the active submode directly depends on a condition:

$$(C.26) \quad \text{condBraking} \equiv \text{brakingRequired} \vee v > v_{\text{allowed}}$$

In case of enforced braking, the submode `braking` defines  $a$  to be the maximum deceleration until the train stops. Otherwise, there is a distinction between unrestricted and restricted appliance of  $a_{\text{user}}$ : The mode `restricted` guarantees that the minimum (0) and maximum ( $v_{\text{allowed}}$ ) velocities are not violated, else `unrestricted` maintains  $a = a_{\text{user}}$ . Again, a condition controls this:

$$(C.25) \quad \text{condUnrestricted} \equiv (v < v_{\text{allowed}} \vee a_{\text{user}} < 0) \wedge (v > 0 \vee a_{\text{user}} > 0)$$

Finally, mode `braking` is re-used in `movementControl` – if the signal `emergency` (that is caused by violating a velocity target point) is received, the train is definitely stopped.

**Control Point.** For the entry and exit of modes, the notion of *control points* is introduced. Each mode has a default entry and a default exit, which is graphically represented by the mode’s border line. Exiting via the default exit automatically preserves the history of the mode, i.e. the information about which submode instance was active, and entering via the default entry then automatically restores the mode instance’s history. In the examples presented here (Fig. 1.13, Fig. 1.14), solely default control points are used. In order to bypass the automatic history, explicit entry and exit points can be modeled, such that on entry, no history is evaluated, and on exit, the history is cleared. Within the crossing controller’s specification (given in appendix C.2), explicit control points are used, for example in Fig. C.76 on page 277.

**Control Point Instance.** Corresponding to the distinction of modes and mode instances, separate control point instances represent control points for different instances of a mode.

**Transition.** A transition models a discrete step of the state-machine. It connects two modes (via control points, or control point instances, respectively), and transfers control on activation from the source mode to the target mode. A transition can be labeled with (1) a trigger expression, (2) a guard expression, and (3) a sequence of action expressions. The guard expression is a boolean expression that always decides whether the transition may be activated or not. Without a trigger expression, the transition is enabled, whenever control resides in its source mode, and the guard is satisfied. For example, if the mode `BrakingNotRequired` in Fig. 1.13 has currently control, and the condition (C.28) `condBrakingRequired` is satisfied, then the transition on the upper left is enabled and can fire, but need not, and therefore is not urgent.<sup>4</sup>

<sup>4</sup>Here, urgency is *modeled* because the invariant constraint of `BrakingNotRequired` is complementary.

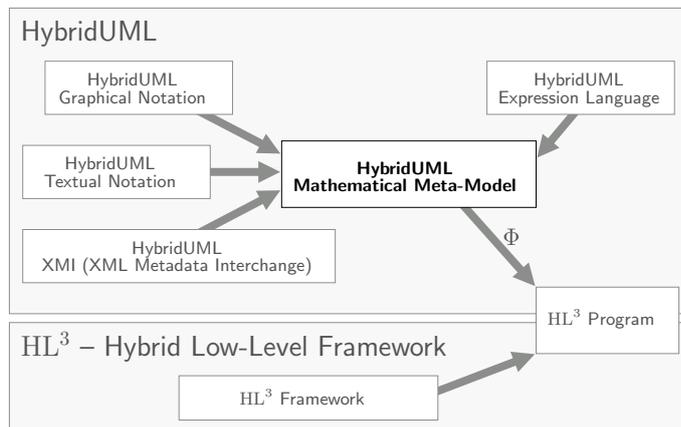
Additionally, a trigger expression defines a signal that must be active, such that the transition is enabled. When the signal is active, then the transition is automatically urgent, in contrast to transitions without trigger specifications. The behavior of `MovementController` (Fig. 1.14) always reacts to the signal *emergency* by a transition to mode `braking`. Since no explicit guard is specified, it is implicitly true.

Apart from the transfer of control to the target mode, the sequence of discrete actions which are attached to a transition is executed. For example, on switching from `BrakingNotRequired` to `BrakingRequired`, the `BrakePointController` (Fig. 1.13) assigns `true` to its variable `brakingRequired`. Note that apart from assignments, the raising of signals can be done with the sequence actions (see Fig. C.38 on page 249, for example).



## Chapter 2

# HybridUML Mathematical Meta-Model



This chapter defines the starting point for the transformation of HybridUML models to their executable semantics. Instead of using the graphical HybridUML syntax directly, we define a data structure that exactly represents HybridUML models syntactically, but non-graphically. In terms of UML, this data structure is the HybridUML meta-model.

This separation of the meta-model from its graphical representation is the usual UML approach. The benefits are that (1) the meta-model is directly usable for transformation  $\Phi$ , and that (2) the HybridUML semantics is independent from the graphical notation.

This chapter defines a syntax for HybridUML specifications – the Mathematical Meta-Model, which is the HybridUML meta-model in terms of UML. In contrast to the illustration given in section 1.3, the syntax is defined non-graphically, but formally in the mathematical sense. The main reason for this is that the focus of this thesis is on the transformation  $\Phi$  from HybridUML specifications into an executable system, which is defined formally itself. The syntax is therefore given in a form that is appropriate as input for the transformation.

Further, the separation of the meta-model from its graphical representation is the usual UML approach. As a consequence, the transformation semantics of HybridUML models is independent from the graphical representation.

The correlation between the Mathematical Meta-Model and the graphical notation of section 1.3 is straightforward, therefore it is not given explicitly in this thesis.

The initial HybridUML presentation [BBHP03] applies the standard UML 2.0 procedure to define a specialization of UML – the *profile mechanism*. HybridUML is defined as a profile, i.e. the UML meta-model is modified by the application of stereotypes. The profile definition itself is given by means of the Meta Object Facility (MOF) [OMGO03], which is the meta-model language of UML.<sup>1</sup>

The profile consists of a mixture of graphical notations, OCL expressions, and natural language. This is the recommended approach to adapt UML, but the resulting profile is not mathematically formal. An additional drawback is that the resulting meta-model is at least as large as the original UML meta-model, since all modifications are realized by the *addition* of stereotypes (which include constraints, textual descriptions, etc.).

Therefore, we define the Mathematical Meta-Model, i.e. the HybridUML meta-model explicitly, using mathematical definitions. The Mathematical Meta-Model is intended to be equivalent to the profile definition to a large extent. However, in addition to small technical differences, the concept of event-based communication is integrated.

In order to make it more comprehensible, the Mathematical Meta-Model is augmented by descriptions of the intended purpose of the respective entities, i.e. *intuitive semantics* are given.

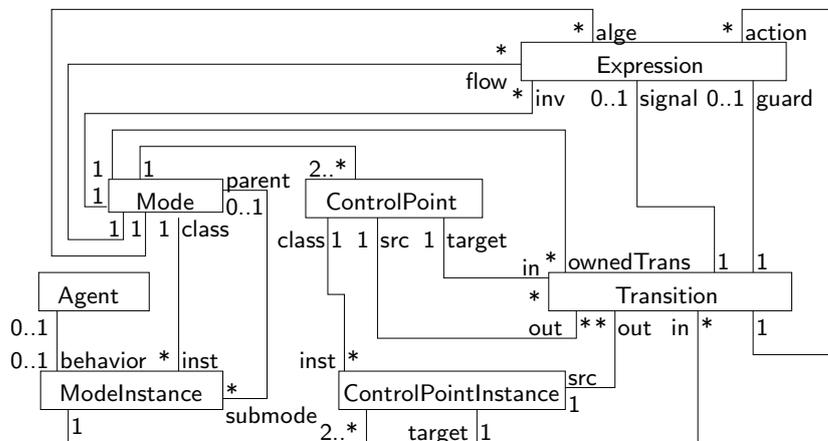


Figure 2.1: HL<sup>3</sup> Part of the Mathematical Meta-Model that provides the behavioral specification, illustrated as UML class diagram.

**HybridUML Specifications.** A HybridUML specification is a tuple

$$spec = (A, AI, a_{itl}, V, \sigma_V, S, P_V, PI_V, C_V, P_S, PI_S, C_S, DT, M, MI, CP, CPI, T, Exp)$$

<sup>1</sup>The MOF features used for the UML meta-model constitute roughly a subset of UML itself.

with a set of agents  $A$ , a set of agent instances  $AI$ , a dedicated top-level agent instance  $ai_{tl} \in AI$ , a set of properties  $V$ , a set of property values  $\sigma_V$ , a set of signals  $S$ , a set of variable ports  $P_V$ , a set of variable port instances  $PI_V$ , a set of variable connectors  $C_V$ , a set of signal ports  $P_S$ , a set of signal port instances  $PI_S$ , a set of signal connectors  $C_S$ , a set of datatypes  $DT$ , a set of modes  $M$ , a set of mode instances  $MI$ , a set of control points  $CP$ , a set of control point instances  $CPI$ , a set of transitions  $T$ , and a set of expressions  $Exp$ .

## 2.1 Structural Specification

In this section, the part of HybridUML specifications that defines the *structure* of the system is given. This contains everything that is represented in *class diagrams* and *composite structure diagrams*, as it was illustrated in section 1.3.

**Agent.** Agents are the main structural building block of a HybridUML specification. An Agent is a *class* in the usual sense – it represents a set of similar objects, which contain data as well as (optional) behavior. These objects make up an object tree, in that each object either (1) forms a part of the *structural hierarchy* of the system, or (2) provides the context for a state machine which defines sequential *behavior*. The latter are the leafs of the object tree. Technically, an agent is either a *basic agent* or a *composite agent*. A basic agent has exactly one state machine called *mode* that defines its behavior, but has no parts (i.e. contained agent instances). A composite agent contains parts, but has no own behavior:

$$\begin{aligned} behavior_A &: A \rightarrow \mathcal{P}(MI) \\ part_A &: A \rightarrow \mathcal{P}(AI) \\ \forall a \in A \bullet \\ &part_A(a) = \emptyset \Leftrightarrow |behavior_A(a)| = 1 \\ &\wedge part_A(a) \neq \emptyset \Leftrightarrow behavior_A(a) = \emptyset \end{aligned}$$

Agents contain variables, parameters, and signals:

$$\begin{aligned} var_A &: A \rightarrow \mathcal{P}(V) \\ param_A &: A \rightarrow \mathcal{P}(V) \\ sig_A &: A \rightarrow \mathcal{P}(S) \end{aligned}$$

Variables are the objects' variables in the usual sense.<sup>2</sup>

Parameters are special private and read-only variables with a multiplicity of one. They are distinct from other variables in that they are available during the construction of the system's static structure, rather than afterwards.

$$\begin{aligned} \forall a \in A \bullet param_A(a) &\subseteq var_A(a) \\ \forall a \in A, v \in V \bullet v \in param_A(a) &\Rightarrow vis_V(v) = priv \\ \forall a \in A, v \in V \bullet v \in param_A(a) &\Rightarrow acc_V(v) = ro \\ \forall a \in A, v \in V \bullet v \in param_A(a) &\Rightarrow mult_V(v) = 1 \end{aligned}$$

---

<sup>2</sup>There are different terms which are commonly used for *variables* within classes, at least *attribute*, *property*, or *member*. We will use them synonymously in this thesis.

There is a special “ID” parameter for each agent:

$$\forall a \in A \bullet v_{id,a} \in param_A(a)$$

Signals are incidents without time duration.<sup>3</sup> Basic agents can synchronize by sending or receiving signals, respectively. The sending (or raising) of a signal is non-blocking: Raised signals are multicasted immediately to all agents that potentially receive the signal. The current internal state of each basic agent determines, whether the signal is actually received or silently lost.

Each variable, parameter, and signal is exclusively contained by an agent, i.e.:

$$\begin{aligned} \forall v \in V, a_1, a_2 \in A \bullet \\ & (v \in var_A(a_1) \wedge v \in var_A(a_2) \Rightarrow a_1 = a_2) \\ \forall p \in V, a_1, a_2 \in A \bullet \\ & (p \in param_A(a_1) \wedge p \in param_A(a_2) \Rightarrow a_1 = a_2) \\ \forall s \in S, a_1, a_2 \in A \bullet \\ & (s \in sig_A(a_1) \wedge s \in sig_A(a_2) \Rightarrow a_1 = a_2) \end{aligned}$$

An agent may have ports for its variables and signals, in order to share them with its environment as well as with its contained subagent instances:

$$\begin{aligned} port_{A,Var} : A \rightarrow \mathcal{P}(P_V) \\ port_{A,Sig} : A \rightarrow \mathcal{P}(P_S) \end{aligned}$$

Each port represents a variable or signal of the agent:

$$\begin{aligned} \forall p \in P_V, a \in A \bullet \\ & p \in port_{A,Var}(a) \Rightarrow var_{P_V}(p) \in var_A(a) \\ \forall p \in P_S, a \in A \bullet \\ & p \in port_{A,Sig}(a) \Rightarrow sig_{P_S}(p) \in sig_A(a) \end{aligned}$$

As with variables and signals, the ports are exclusively contained by one agent each:

$$\begin{aligned} \forall p \in P_V, a_1, a_2 \in A \bullet \\ & (p \in port_{A,Var}(a_1) \wedge p \in port_{A,Var}(a_2) \Rightarrow a_1 = a_2) \\ \forall p \in P_S, a_1, a_2 \in A \bullet \\ & (p \in port_{A,Sig}(a_1) \wedge p \in port_{A,Sig}(a_2) \Rightarrow a_1 = a_2) \end{aligned}$$

If an agent is composite, it can have variable and signal connectors. Connectors connect ports and port instances, in order to define shared variables and shared signals inside the agent.

$$\begin{aligned} conn_{A,Var} : A \rightarrow \mathcal{P}(C_V) \\ conn_{A,Sig} : A \rightarrow \mathcal{P}(C_S) \\ \forall a \in A \bullet \\ & part_A(a) = \emptyset \Rightarrow conn_{A,Var}(a) = \emptyset \wedge conn_{A,Sig}(a) = \emptyset \end{aligned}$$

---

<sup>3</sup>A more common term is *event*, but for UML conformity, we prefer *signal*.

Each agent has a set of initial state constraints. These constraints must hold for each of the agent's objects after construction of the static structure of the system. Otherwise, there is no defined execution of the complete system.

$$initState_A : A \rightarrow \mathcal{P}(Exp)$$

**AgentInstance.** An agent instance represents a subset of the objects of a dedicated agent. Therefore, agent instances are mapped to the corresponding agent:

$$agent_{AI} : AI \rightarrow A$$

A multiplicity expression defines the number of objects that are represented:

$$mult_{AI} : AI \rightarrow Exp$$

The multiplicity has to be at least one. If the agent instance is defined within the context of a parent agent (which is almost always the case), then the number of objects is further multiplied by the number of duplicates of the parent agent.

For connecting the agent instance's properties or signals with its environment, the corresponding variable or signal ports of the corresponding agent are instantiated for the agent instance:

$$portIns_{AI,Var} : AI \rightarrow \mathcal{P}(PI_V)$$

$$portIns_{AI,Sig} : AI \rightarrow \mathcal{P}(PI_S)$$

Of course, no other ports can be instantiated, i.e.:

$$\forall pi \in PI_V, ai \in AI \bullet$$

$$pi \in portIns_{AI,Var} \Rightarrow port_{PI_V}(pi) \in port_{A,Var}(agent_{AI}(ai))$$

$$\forall pi \in PI_S, ai \in AI \bullet$$

$$pi \in portIns_{AI,Sig} \Rightarrow port_{PI_S}(pi) \in port_{A,Sig}(agent_{AI}(ai))$$

Each port instance is connected to exactly one agent instance:

$$\forall pi \in PI_V, ai_1, ai_2 \in AI \bullet$$

$$(pi \in portIns_{AI,Var}(ai_1) \wedge pi \in portIns_{AI,Var}(ai_2)) \Rightarrow ai_1 = ai_2$$

$$\forall pi \in PI_S, ai_1, ai_2 \in AI \bullet$$

$$(pi \in portIns_{AI,Sig}(ai_1) \wedge pi \in portIns_{AI,Sig}(ai_2)) \Rightarrow ai_1 = ai_2$$

Each parameter of the agent gets exactly one value specification for the agent instance. This is determined and assigned during the creation of the static structure of the system.

$$paramVal_{AI} : AI \rightarrow \mathcal{P}(\sigma_V)$$

$$\forall ai \in AI \bullet \forall p \in param_A(agent_{AI}(ai)) \bullet$$

$$|\{v \in paramVal_{AI}(ai) \mid var_{\sigma_V}(v) = p\}| = 1$$

The initial state space can be constrained on agent instance level in the same way as on agent level:

$$initState_{AI} : AI \rightarrow \mathcal{P}(Exp)$$

The conjunction of all initial state constraints from the agent instance as well as from the corresponding agent defines the initial state space for the agent instance's objects.

**Property.** Properties (alias variables<sup>2</sup>) of agents are typed:

$$type_V : V \rightarrow DT$$

Each variable has a multiplicity, a visibility, and an access policy:

$$mult_V : V \rightarrow Exp$$

$$vis_V : V \rightarrow \{priv, pub\}$$

$$acc_V : V \rightarrow \{ro, rw\}$$

Every variable is implicitly an array, such that there are  $n \geq 1$  copies of it. This is defined by its multiplicity. The visibility is either *public*, i.e. the variable can be connected to variables of other agents, or *private* to prevent this. The access policy defines whether the variable is writable or constant.

**DataType.** Different kinds of datatypes are available for HybridUML – (1) primitive datatypes, (2) structured datatypes and (3) enumeration types:

$$kind_{DT} : DT \rightarrow \{prim, struc, enum\}$$

$$DT_{prim} = \{t \in DT \mid kind_{DT}(t) = prim\}$$

$$DT_{struc} = \{t \in DT \mid kind_{DT}(t) = struc\}$$

$$DT_{enum} = \{t \in DT \mid kind_{DT}(t) = enum\}$$

The primitive types are predefined, these are (1) *boolean*, representing values  $b \in \mathbb{B}$ , (2) *integer*, representing values  $i \in \mathbb{Z}$ , and (3) *real*, which represents values  $r \in \mathbb{R}$ . Additionally, the type (4) *analog real* is used to distinguish variables which can be modified continuously from discrete real-valued variables.

$$\forall t \in DT \bullet$$

$$kind_{DT}(t) = prim \Rightarrow t \in \{bool, int, real, anaReal\}$$

Enumeration types are defined within the HybridUML model. They have exclusive literals, i.e. each enumeration literal is mapped to exactly one enumeration type. There are no other data types with enumeration literals.

$$dt_L : L \rightarrow DT$$

$$lit_{DT} : DT \rightarrow \mathcal{P}(L)$$

$$t \mapsto \{l \in L \mid dt_L(l) = t\}$$

$$\forall t \in DT \bullet$$

$$kind_{DT}(t) \neq enum \Rightarrow lit_{DT}(t) = \emptyset$$

Structured data types are also defined within the model. Each structured data type contains a list of properties:

$$varseq_{DT} : DT \rightarrow seq V$$

The unsorted shorthand notation for structured data types is:

$$var_{DT} : DT \rightarrow \mathcal{P}(V)$$

$$t \mapsto \{v \in V \mid \exists i \in \mathbb{N} \bullet (i, v) \in varseq_{DT}(t)\}$$

Only structured data types contain properties:

$$\begin{aligned} \forall t \in DT \bullet \\ \text{kind}_{DT}(t) \neq \text{struc} \Rightarrow \text{var}_{DT}(t) = \emptyset \end{aligned}$$

Properties within structured data types are unique:

$$\begin{aligned} \forall p \in V, t \in DT, i_1, i_2 \in \mathbb{N} \bullet \\ ((i_1, p) \in \text{varseq}_{DT}(t) \wedge (i_2, p) \in \text{varseq}_{DT}(t) \Rightarrow i_1 = i_2) \\ \forall p \in V, t_1, t_2 \in DT \bullet \\ (p \in \text{var}_{DT}(t_1) \wedge p \in \text{var}_{DT}(t_2) \Rightarrow t_1 = t_2) \end{aligned}$$

Properties are either part of a structured data type, or part of an agent:

$$\begin{aligned} \forall p \in V, t \in DT \bullet \\ p \in \text{var}_{DT}(t) \Rightarrow \exists a \in A \bullet (p \in \text{var}_A(a) \vee p \in \text{param}_A(a)) \\ \forall p \in V, a \in A \bullet \\ (p \in \text{var}_A(a) \vee p \in \text{param}_A(a)) \Rightarrow \exists t \in DT \bullet p \in \text{var}_{DT}(t) \end{aligned}$$

**PropertyValue.** Properties within the HybridUML model can have a value specification. This value specification determines the initial value of the property. For constant properties, particularly for parameters, the initial value thus is the constant value of the property. It is given by an expression:

$$\text{exp}_{\sigma_V} : \sigma_V \rightarrow \text{Exp}$$

Each property value specification is mapped to its property:

$$\text{var}_{\sigma_V} : \sigma_V \rightarrow V$$

There are no values for the special “ID” parameters:

$$\forall a \in A \bullet \exists \text{val} \in \sigma_V \bullet \text{var}_{\sigma_V}(\text{val}) = v_{id,a}$$

**VariablePort.** A variable port provides a property, such that the property can be connected with other properties, by means of variable connectors. Each variable port represents exactly one property:

$$\text{var}_{P_V} : P_V \rightarrow V$$

The property must be publicly visible:

$$\forall p \in P_V \bullet \text{vis}_V(\text{var}_{P_V}(p)) = \text{pub}$$

The variable port has a separate access policy that can restrict the access of a read/write variable to read-only:

$$\text{acc}_{P_V} : P_V \rightarrow \{\text{ro}, \text{rw}\}$$

Read/write ports for read-only properties are not allowed:

$$\forall p \in P_V \bullet \text{acc}_V(\text{var}_{P_V}(p)) = \text{ro} \Rightarrow \text{acc}_{P_V}(p) = \text{ro}$$

**VariablePortInstance.** Variable port instances correspond to variable ports. Each variable port instance represents a variable port for a particular agent instance. Thus, variable port instances are mapped to their variable port:

$$port_{PI_V} : PI_V \rightarrow P_V$$

There can be several variable port instances that correspond to a variable port, particularly a variable port instance may represent only a subset of the indices which are defined by the property's multiplicity. An expression defines the set of indices which are represented by the variable port instance:

$$indices_{PI_V} : PI_V \rightarrow Exp$$

**VariableConnector.** A variable connector connects an arbitrary number of variable port instances and up to one variable port:

$$portIns_{C_V} : C_V \rightarrow \mathcal{P}(PI_V)$$

$$port_{C_V} : C_V \rightarrow \mathcal{P}(P_V)$$

$$\forall c \in C_V \bullet |port_{C_V}(c)| \leq 1$$

The data types of the port instances and of the port must match. This ensures that the resulting shared variable has a well-defined type.

$$\forall c \in C_V \bullet \forall p_1, p_2 \in port_{PI_V}(portIns_{C_V}(c)) \cup port_{C_V}(c) \bullet$$

$$type_V(var_{P_V}(p_1)) = type_V(var_{P_V}(p_2))$$

All connected port instances and the port must be local to the agent that owns the connector. The port instances are from the contained subagent instances and therefore connect their properties. The (optional) local port is a port of the agent itself, and thus make the internally connected variables externally available for connection, through agent instances of this agent.

$$\forall c \in C_V, a \in A \bullet c \in conn_{A, Var}(a) \Rightarrow$$

$$((\forall p \in port_{C_V}(c) \bullet p \in port_{A, Var}(a))$$

^

$$(\forall pi \in portIns_{C_V}(c), ai \in AI \bullet$$

$$pi \in portIns_{AI, Var}(ai) \Rightarrow ai \in part_A(a)))$$

There are different kinds of connectors:

$$kind_{C_V} : C_V \rightarrow \{ptp, mult\}$$

They differ in how indices of variables are mapped. A *point-to-point* connector matches each index individually and thus acts as a set of separate connectors. A *multicast* connector acts as a single connector that connects every two indices of every two variables.

**Signal.** HybridUML signals are themselves not typed, but have a parameter list that is defined by a list of data types. Signals are always processed at discrete points in time, therefore the special type *anaReal* is inappropriate here.

$$paramTypes_S : S \rightarrow seq(DT \setminus \{anaReal\})$$

Similarly to properties, a signal specification defines an array of  $n \geq 1$  copies. Therefore, signals also have a multiplicity:

$$mult_S : S \rightarrow Exp$$

**SignalPort.** A signal port provides a signal for connection with other signals, in the same way as variable ports provide properties. Each signal port acts for one signal:

$$sig_{P_S} : P_S \rightarrow S$$

The signal port has an access policy that defines whether the signal can be sent or received through it:

$$acc_{P_S} : P_S \rightarrow \{recv, send\}$$

In contrast to variable ports, signal ports control the access to signals exclusively, since signals do not have access policies themselves. However, there must be no signal that is received *and* sent, therefore all ports of a signal have the same access policy:

$$\forall p_1, p_2 \in P_S \bullet sig_{P_S}(p_1) = sig_{P_S}(p_2) \Rightarrow acc_{P_S}(p_1) = acc_{P_S}(p_2)$$

**SignalPortInstance.** Signal port instances correspond to signal ports, in the same way that variable port instances correspond to variable ports:

$$port_{PI_S} : PI_S \rightarrow P_S$$

According to variable port instances, an expression defines the set of the port indices which are represented by this port instance:

$$indices_{PI_S} : PI_S \rightarrow Exp$$

**SignalConnector.** A signal connector connects an arbitrary number of signal port instances and up to one signal port:

$$portIns_{C_S} : C_S \rightarrow \mathcal{P}(PI_S)$$

$$port_{C_S} : C_S \rightarrow \mathcal{P}(P_S)$$

$$\forall c \in C_S \bullet |port_{C_S}(c)| \leq 1$$

The parameter lists of the signals which are represented by the port instances and of the port must match, such that a signal and particularly its actual parameters can be successfully transmitted over the connector:

$$\forall c \in C_S \bullet \forall p_1, p_2 \in port_{PI_S}(portIns_{C_S}(c)) \cup port_{C_S}(c) \bullet \\ paramTypes_S(sig_{P_S}(p_1)) = paramTypes_S(sig_{P_S}(p_2))$$

As with variable connectors, all connected port instances and the port must be local to the agent that owns the connector, in order to connect port instances from the contained subagent instances internally, and to optionally provide them for external connection:

$$\forall c \in C_S, a \in A \bullet c \in conn_{A, sig}(a) \Rightarrow \\ ((\forall p \in port_{C_S}(c) \bullet p \in port_{A, sig}(a)) \\ \wedge \\ (\forall pi \in portIns_{C_S}(c), ai \in AI \bullet \\ pi \in portIns_{AI, sig}(ai) \Rightarrow ai \in part_A(a)))$$

The kinds of connectors coincide with variable connector kinds, such that point-to-point and multicast connections are distinguished for signal connectors, too:

$$kind_{C_S} : C_S \rightarrow \{ptp, mult\}$$

## 2.2 Behavioral Specification

This section provides the part of the HybridUML specification syntax that defines the system's *behavior*. Graphically, the behavior was specified by *statechart diagrams* in section 1.3.

**Mode.** A mode acts as a definition of behavior. Modes are hierarchical *state machines*, in that they can contain submodes. In contrast to plain UML, each submode is always a state machine itself, therefore we do not distinguish between *state machine* and *state*. Further, we prefer the term *mode*, since for hybrid systems, a state usually describes the combination of *control state* and *data state*, the latter including the current point in time. Therefore, residing in a specific control state for a positive time duration involves a *set of states*, which are subsumed to a so-called *mode*.<sup>4</sup> In order to create the behavioral hierarchy, a mode may contain submode instances:

$$\text{submode}_M : M \rightarrow \mathcal{P}(MI)$$

Every submode instance is at most part of one mode:

$$\begin{aligned} \forall mi \in MI, m_1, m_2 \in M \bullet \\ (mi \in \text{submode}_M(m_1) \wedge mi \in \text{submode}_M(m_2) \Rightarrow m_1 = m_2) \end{aligned}$$

Discrete behavior is modeled by transitions, which conceptually transfer control from a currently active mode to a new mode. Technically, modes are equipped with *control points* which act as exit or entry for transitions:

$$\text{cp}_M : M \rightarrow \mathcal{P}(CP)$$

Each control point is exclusively contained by one mode:

$$\begin{aligned} \forall c \in CP, m_1, m_2 \in M \bullet \\ (c \in \text{cp}_M(m_1) \wedge c \in \text{cp}_M(m_2) \Rightarrow m_1 = m_2) \end{aligned}$$

There are two special control points for each mode – *default entry* and *default exit*:

*Default Exit* Transitions starting from the default exit point can fire independently of the mode's internal state, and thus are so-called *group transitions*.<sup>5</sup> In contrast, transitions starting from other exit points can only be taken, if control is explicitly transferred to that exit point before.

*Default Entry* Transitions entering the default entry point resume the *history* of that mode, i.e. the submodes which were active before a preceding interrupt are reactivated. If no history is available, an initialization transition activates an initial mode. However, mode entry via normal entry points always requires explicit control transfer to a submode.

<sup>4</sup>These terms particularly comply to [AGLS01]. In [Hen96], instead of *mode* the term *control mode* is used.

<sup>5</sup>Group transitions are called *interrupt transitions*, alternatively.

$$\begin{aligned}
de_M &: M \rightarrow CP \\
\forall m \in M \bullet (de_M(m) \in cp_M \wedge kind_{CP}(de_M(m)) = entry) \\
dx_M &: M \rightarrow CP \\
\forall m \in M \bullet (dx_M(m) \in cp_M \wedge kind_{CP}(dx_M(m)) = exit)
\end{aligned}$$

A mode then can contain transitions that connect its own control points and control points of submodes. Transitions between submodes switch control within the mode, whereas transitions to or from the mode itself prepare the loss of control or handle the gain of control of the mode, respectively.

$$trans_M : M \rightarrow \mathcal{P}(T)$$

No transition is contained by more than one mode:

$$\begin{aligned}
\forall t \in T, m_1, m_2 \in M \bullet \\
(t \in trans_M(m_1) \wedge t \in trans_M(m_2)) \Rightarrow m_1 = m_2
\end{aligned}$$

Continuous behavior is given by algebraic and flow constraints that define continuous evolutions of variables when the mode is active. Both are given by expressions, see chapter 3 for details on them, including the distinction between them.

$$\begin{aligned}
flow_M &: M \rightarrow \mathcal{P}(Exp) \\
alge_M &: M \rightarrow \mathcal{P}(Exp)
\end{aligned}$$

Invariant constraints define, whether the mode may be currently active or not. This can particularly disable the continuous behavior and thus enforce discrete behavior. Technically, invariants are given by boolean expressions, which are described in chapter 3.

$$inv_M : M \rightarrow \mathcal{P}(Exp)$$

**ModeInstance.** A mode instance is a concrete occurrence of a mode. The discrimination of modes and mode instances is provided solely for re-use of mode specifications. In the graphical examples of section 1.3, modes are mostly defined implicitly along with a single mode instance specification. An example of separate mode and mode instance definition is given in Fig. 1.14 and Fig. 1.15.

$$mode_{MI} : MI \rightarrow M$$

Accordingly, the mode's control points are also instantiated. For each control point of the mode, exactly one control point instance exists for the mode instance:

$$\begin{aligned}
cpi_{MI} &: MI \rightarrow \mathcal{P}(CPI) \\
\forall mi \in MI \bullet \forall c \in cp_M(mode_{MI}(mi)) \bullet \\
&\quad \exists ci \in cpi_{MI}(mi) \bullet cp_{CPI}(ci) = c \\
\forall mi \in MI \bullet \forall ci_1, ci_2 \in cpi_{MI}(mi) \bullet \\
&\quad (cp_{CPI}(ci_1) = cp_{CPI}(ci_2)) \Rightarrow ci_1 = ci_2 \\
\forall mi \in MI, ci \in CPI \bullet \\
&\quad ci \in cpi_{MI}(mi) \Rightarrow cp_{CPI}(ci) \in cp_M(mode_{MI}(mi))
\end{aligned}$$

A control point instance is at most part of one mode instance:

$$\begin{aligned} \forall ci \in CPI, mi_1, mi_2 \in MI \bullet \\ (ci \in cpi_{MI}(mi_1) \wedge ci \in cpi_{MI}(mi_2) \Rightarrow mi_1 = mi_2) \end{aligned}$$

**ControlPoint.** Control points define the entries to modes and exits from modes. *Exit points* act as sources of transitions, whereas *entry points* act as transition targets.

$$kind_{CP} : CP \rightarrow \{entry, exit\}$$

**ControlPointInstance.** A control point instance is a concrete occurrence of a control point and is part of a mode instance:

$$cp_{CPI} : CPI \rightarrow CP$$

**Transition.** A transition represents a discrete behavioral step. It implicitly connects mode instances or a mode and a submode instance, by connecting their control point instances or a control point with a control point instance. Three combinations are allowed:

- (1) Connecting an entry point  $e$  of mode  $m$  with an entry point instance  $e_{sub}$  of a submode instance  $m_{sub}$ : When control is transferred to  $m$  via  $e$ , then  $m_{sub}$  gains control through  $e_{sub}$ .
- (2) Connecting an exit point instance  $x_1$  of a submode  $m_1$  with an entry point instance  $e_2$  of a submode  $m_2$ : When the transition fires, control is transferred from  $m_1$  to  $m_2$ .
- (3) Connecting an exit point instance  $x_{sub}$  of a submode instance  $m_{sub}$  with a non-default exit point  $x$  of the parent mode  $m$ : When the transition fires, control is transferred to  $x$ , such that outgoing transitions from  $x$  can fire and  $m$  loses control.

Note that transitions thus are not only used to switch control *between* modes, but also determine how control is assigned *within* modes.

$$\begin{aligned} src_T : T \rightarrow \{CP \cup CPI\} \\ tar_T : T \rightarrow \{CP \cup CPI\} \\ \forall t \in T \bullet \\ \quad src_T(t) \in CP \wedge kind_{CP}(src_T(t)) = entry \\ \quad \wedge tar_T(t) \in CPI \wedge kind_{CP}(cp_{CPI}(tar_T(t))) = entry \\ \vee \\ \quad src_T(t) \in CPI \wedge kind_{CP}(cp_{CPI}(src_T(t))) = exit \\ \quad \wedge tar_T(t) \in CPI \wedge kind_{CP}(cp_{CPI}(tar_T(t))) = entry \\ \vee \\ \quad src_T(t) \in CPI \wedge kind_{CP}(cp_{CPI}(src_T(t))) = exit \\ \quad \wedge tar_T(t) \in CP \wedge kind_{CP}(tar_T(t)) = exit \end{aligned}$$

If either source or target is a control point, then the control point must be contained by the same mode as the transition itself:

$$\forall t \in T, m \in M \bullet$$

$$\begin{aligned}
& src_T(t) \in CP \wedge t \in trans_M(m) \Rightarrow src_T(t) \in cp_M(m) \\
& \forall t \in T, m \in M \bullet \\
& \quad tar_T(t) \in CP \wedge t \in trans_M(m) \Rightarrow tar_T(t) \in cp_M(m)
\end{aligned}$$

If either source or target is a control point instance, then the control point instance must be contained by a submode instance of the mode that contains the transition:

$$\begin{aligned}
& \forall t \in T, m \in M, mi \in MI \bullet \\
& \quad t \in trans_M(m) \wedge src_T(t) \in cpi_{MI}(mi) \Rightarrow mi \in submode_M(m) \\
& \forall t \in T, m \in M, mi \in MI \bullet \\
& \quad t \in trans_M(m) \wedge tar_T(t) \in cpi_{MI}(mi) \Rightarrow mi \in submode_M(m)
\end{aligned}$$

The target of a transition must not be a default exit point:

$$\begin{aligned}
& \forall t \in T, m \in M \bullet \\
& \quad tar_T(t) \in CP \wedge t \in trans_M(m) \Rightarrow tar_T(t) \neq dx_M(m)
\end{aligned}$$

A transition can be equipped with a trigger, which is the specification of a signal to be received. If the signal is currently active (and the guard expression is satisfied, and the source control point has control), then the transition (or a concurrently enabled one) *must* be taken.

$$\begin{aligned}
& sig_T : T \rightarrow \mathcal{P}(Exp) \\
& \forall t \in T \bullet \\
& \quad |sig_T(t)| \leq 1
\end{aligned}$$

A transition can have a guard, which is the specification of a boolean expression. If the guard is satisfied (and the source control point has control), and either (1) there is no trigger specification or (2) the trigger's signal is active, then the transition is enabled. In absence of a trigger, the transition *can* be taken. Particularly, as long as the invariant constraints of the source mode instance are fulfilled, the system may reside in that mode instance and may continue continuous behavior.

$$\begin{aligned}
& grd_T : T \rightarrow \mathcal{P}(Exp) \\
& \forall t \in T \bullet \\
& \quad |grd_T(t)| \leq 1
\end{aligned}$$

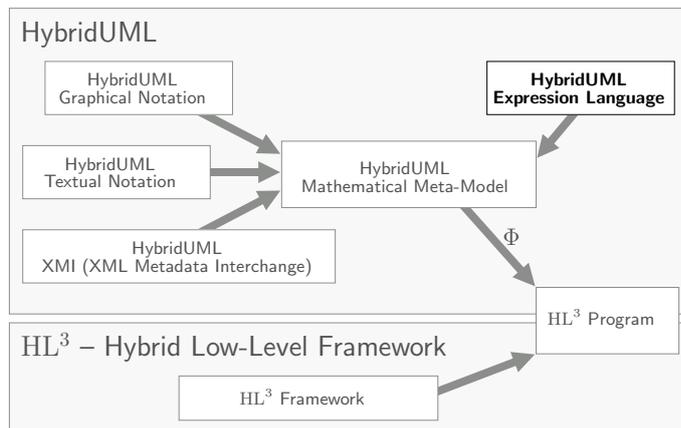
Each transition has a list of actions to be taken when the transition fires. Actions, as well as triggers and guards, are given by expressions and are discussed in chapter 3.

$$act_T : T \rightarrow seq\ Exp$$



## Chapter 3

# HybridUML Expression Language



In this chapter, the HybridUML Expression Language is defined. It defines the expressions that can be used within a HybridUML model, e.g. boolean expressions from mode invariants or transition conditions, or assignment expressions from transition actions.

The syntax of the HybridUML Expression Language is given, and its intuitive semantics is explained informally. An *intermediate semantics* is given, which is a data structure that will be used to define the formal semantics. This will be given in chapter 5, since it depends on the transformation  $\Phi$  of the complete HybridUML model to its executable semantics.

The HybridUML Expression Language (HybEL) defines the syntax of expressions  $exp \in Exp_{HybEL}$  that can be used within HybridUML specifications. The diagrammatic part of the specification embeds textual parts which define constraints and assignments for the behavior as well as for the structure of the system.

HybridUML is not restricted to be used with the HybridUML Expression Language; the application of different expression languages is possible. Syntactically, the set  $Exp$  of expressions contained in a given HybridUML specification

can be any set of expressions according to an arbitrary expression language. Semantically, however, the choice of expression language has a heavy impact on the transformation  $\Phi$  defined in this thesis. For any HybridUML specification with arbitrary expressions (of the chosen language), the transformation explicitly defines its semantics. This directly depends on the chosen expression language.

Therefore, HybEL is customized as a small expression language that fits HybridUML, such that

- it is expressive enough for the design of embedded applications,
- it supports HybridUML's main concepts, particularly the calculation of time-continuous values,
- and that it is concise in order to facilitate a comprehensible semantics by transformation.

The most obvious alternative is the Object Constraint Language (OCL) [OMGc], which is UML's standard expression language. In many respects, OCL is more expressive than HybridUML Expression Language, e.g. it contains powerful collection operations providing set comprehension, as well as universal and existential quantifiers on sets of arbitrary objects. We evaluated the combination of HybridUML and OCL by means of the BART case study, with focus on validation and verification [BZL04]. The applied OCL-based validation and verification concept is described in [Ric02].

However, many features of OCL are useless for HybridUML specifications, such as navigation along associations, since HybridUML excludes a considerable amount of UML features, e.g. associations. In contrast, OCL does not fulfill every demand on an expression language for HybridUML. For example, OCL is explicitly designed to be effect-free and thus does not provide assignment expressions. As a consequence, in order to use OCL with the transformation concept of this thesis, extensive customizations would be required, consisting of restrictions of as well as extensions to the OCL standard. To our opinion, this would reduce the main benefit of using OCL significantly. Therefore, we prefer the custom-made expression language HybEL, and assume  $Exp \subseteq Exp_{HybEL}$  for this thesis.

The rest of this chapter provides the following: (1) The relation of HybEL expressions to a given HybridUML specification is described in section 3.1 – *roles* of expressions are identified, leading to the definition of the *context* of expressions. This implies a conceptual separation of full HybEL expressions and embedded *identifier expressions*. Further it is motivated that HybEL expressions do not have a complete semantics on their own, but only within the complete transformation  $\Phi$  of HybridUML specifications. (2) Section 3.2 provides the syntax and the *intermediate semantics* of identifier expressions, which is a data structure that acts as input for the transformation  $\Phi$ . (3) The syntax and intermediate semantics of full HybEL expressions is given in section 3.3. (4) Finally, section 3.4 defines an incomplete semantics which is used for the syntax of HybEL expressions in section 3.3.1.

For a less detailed overview of the syntax of the HybridUML Expression Language, appendix B contains an EBNF grammar, which omits some of the details given in this chapter.

### 3.1 Context of Expressions

This section defines the *context of expressions*. As a prerequisite, *roles of expressions* within a given HybridUML specification are identified. Implicitly, these were already given in chapter 2, but are repeated here explicitly in order to point out the application of expressions within the specification. From this, a list of required language features is determined. The dependencies of expressions on the given HybridUML specification are identified.

**Roles of Expressions.** For use with the *structural* specification part, expressions are attached to properties (including parameters), signals, variable ports, signal ports, agents, and agent instances:

*Initial Property Values* The expression  $exp = exp_{\sigma_V}(val)$  of a property value  $val \in \sigma_V$  is an expression that determines an initial value for the associated property  $v = var_{\sigma_V}(val)$ . The value is calculated from literals, variables, and operations, such that the resulting type is  $t = type_V(v)$ . For properties  $v$  of agents (i.e.  $\exists a \in A \bullet v \in var_A(a)$ ), the expression may refer to variables  $v_p$  of parent agents, i.e. for which holds:  $\exists a, a_p \in A, ai \in AI \bullet (v_p \in var_A(a_p) \wedge ai \in part_A(a_p) \wedge agent_{AI}(ai) = a \wedge v \in var_A(a))$ . In contrast, for properties  $v$  of structured data types (i.e.  $\exists t \in DT_{struc} \bullet v \in var_{DT}(t)$ ), no variables are available.

*Init State Constraints* Constraints on the initial state of the system can be specified by boolean expressions  $exp \in initState_{AI}(ai)$  per agent instance  $ai$  or by boolean expressions  $exp \in initState_A(a)$  per agent  $a$ , in order to model a *set* of admissible start states, rather than a single one.<sup>1</sup> The boolean result is determined from literals, variables, and operations on them. Variables  $v_a$  from the attached agent are accessible, that are  $v_a \in var_A(a)$  or  $v_a \in var_A(agent_{AI}(ai))$ , respectively.

*Property Multiplicities* For each property  $v$  a multiplicity expression  $exp = mult_V(v)$  is given. It defines the number  $n$  of copies for this property, such that an array of  $n$  different values  $v[0]..v[n-1]$  results. Each expression  $exp$  must be of type *int* and must evaluate to a natural number. It is calculated from numerical literals, operations, and variables, which are – for properties of agents, i.e.  $\exists a \in A \bullet v \in var_A(a)$  – the variables  $v_a$  of the agent containing  $v$  itself, i.e. for which hold:  $\exists a \in A \bullet \{v, v_a\} \subseteq var_A(a)$ . No variables are accessible for the calculation of variables of structured data types, i.e. if  $\exists t \in DT_{struc} \bullet v \in var_{DT}(t)$ .

*Signal Multiplicities* For each signal  $s$  a multiplicity expression  $exp = mult_S(s)$  is given, similarly to property multiplicities.

*Agent Instance Multiplicities* An agent instance  $ai$  is equipped with an integer-typed multiplicity expression  $exp = mult_{AI}(ai)$  that defines the number  $n$  of copies that it represents, in the same way that properties and signals are. For the calculation of the integer value, variables  $v_p$  from the parent agent are available – for which hold:  $\exists a_p \in A \bullet (v_p \in var_A(a_p) \wedge ai \in part_A(a_p))$ .

<sup>1</sup>The distinction between “Initial Property Values” and “Init State Constraints” is motivated in chapter 5.

*Property Index Specifications* Since variable port instances  $pi$  may represent a subset instead of all copies of the associated property  $v$ , an expression  $exp = indices_{PI_V}(pi)$  is given that defines a set  $indices$  of integer-typed values. With a property multiplicity  $n$ , the resulting set of represented property copies is  $\{v[i] \mid i \in \{0..n-1\} \cap indices\}$ . For the calculation, literals, operations, and variables from the parent agent can be used. Thus, the variables  $v_p$  with  $\exists a, a_p \in A, ai \in AI \bullet (v_p \in var_A(a_p) \wedge ai \in part_A(a_p) \wedge agent_{AI}(ai) = a \wedge v \in var_A(a))$  can be referred to.

*Signal Index Specifications* Signal index specifications  $exp = indices_{PI_S}(pi)$  are similar to variable index specifications.

Within the *behavioral* specification part, expressions are attached to either modes or transitions. The modes' expressions are:

*Flow Constraints* A flow constraint  $f \in flow_M(m)$  for some mode  $m$  is an expression that is suitable to define a continuous evolution of a variable. Therefore it is an expression that calculates a numeric value and assigns it to a variable  $v$  of type  $type_V(v) = anaReal$ . This can be a conventional assignment, i.e. the variable is directly assigned, and the value is determined from numeric literals, variables, and operations. Alternatively, derivatives of variables can be used either for the calculation or for the assignment, i.e. either for the calculation, a derivative is read, or the value of a variable is assigned indirectly by assigning its derivative.

*Algebraic Constraints* An algebraic constraint  $a \in alge_M(m)$  for some mode  $m$  is, like a flow constraint, an expression that calculates a numeric value and assigns it to a variable  $v$  of type  $type_V(v) = anaReal$ . The difference is that no derivatives can be used with algebraic constraints.

*Invariant Constraints* An invariant constraint  $i \in inv_M(m)$  for some mode  $m$  is an expression that defines iff the mode may be active. It provides a boolean value, which is determined from variables and literals of any type, potentially by the application of suitable operations.

The expressions attached to transitions are:

*Trigger* The trigger  $sig_T(t)$  of a transition  $t$  is an expression that defines exactly one signal  $s$  to be received, in order to enable  $t$ . If the signal carries parameter values, a list of writable variables has to be defined which store these values on signal reception. The variable types must correspond to the signal's parameter definition  $paramTypes_S(s)$ .

*Guard* The guard  $grd_T(t)$  of a transition  $t$  is a boolean expression that enables or disables  $t$ . Similarly to invariant constraints, variables, literals, and operations on them calculate the boolean result. If guard and trigger are both present, their conjunction enables  $t$ .

*Action* The actions  $act_T(t)$  of a transition define a sequence of assignments and signal raise statements that are to be executed when the transition fires. An assignment calculates a new value for some variable  $v$  of any type. The value is given by an expression of appropriate type, which consists of literals, variables, and operations on them. A signal raise statement

defines a signal to be sent. A list of expressions must be provided if the signal is supposed to carry parameter values. The expression types must correspond to the signal's parameter definition  $paramTypes_S(s)$ , such that the signal's parameters are determined.

Because behavioral specifications, i.e. top-level mode instances  $mi$  and all their (recursively) included submode instances and transitions do not have own variables or signals, but are associated with agent instances, they refer to the variables  $v$  and signals  $s$  provided by the corresponding agent instance  $ai$ , i.e.  $behavior_A(agent_{AI}(ai)) = mi \wedge v \in var_A(agent_{AI}(ai))$  or  $behavior_A(agent_{AI}(ai)) = mi \wedge s \in sig_A(agent_{AI}(ai))$ , respectively.

**Required HyBEL Features.** From the possible roles of expressions within the HybridUML specification, a list of elements results that the expression language must provide. From this feature list, the language is developed. The required features are:

1. Literals of all types  $t \in DT$ .
2. Access to variables  $v$  of all types  $type_V(v) \in DT$ . Read and write access are distinguished.
3. Access to derivatives of variables  $v$  of type  $t = anaReal$ . Read and write access are distinguished.
4. Common numeric and boolean operations, like basic arithmetics and comparison of values. Operations depend on the number and type of their arguments.
5. Special operations that define finite sets of integers  $iset \in \mathcal{P}(int)$ .
6. Assignments to writable variables of all types  $t \in DT$ .
7. Assignments to derivatives of writable variables of type  $t = anaReal$ .
8. Access to signals  $s$ . Receive and send access are distinguished.
9. Reception of signals, incl. parameter specifications.
10. Sending of signals, incl. parameter specifications.
11. Index expressions for the determination of variable or signal indices.

**Dependency to the HybridUML Specification.** The available variables and signals within a particular expression depend on the expression's role, and are given by a dedicated agent, which is an associated agent, a parent agent, or an embedding agent. Further, the absence of a corresponding agent leads to an empty set of available variables and signals.

In contrast, the availability of literals, operations, assignments, as well as signal receive and send statements is independent of the embedding specification. Thus, it is exactly the variables and signals that constitute the expression's dependency to the HybridUML specification.

**Context of Expressions.** The *context of expressions* is the set of available variables and signals. Since the access policy of variables and signals is relevant within expressions, but signals themselves have none, it is added explicitly:

$$CTX = \mathcal{P}(V \cup V_{local}) \times \mathcal{P}(S \times \{recv, send\})$$

The contained variables and signals can be accessed by

$$\begin{aligned} var_{CTX} : CTX &\rightarrow \mathcal{P}(V \cup V_{local}) \\ (var, sig) &\mapsto var \\ sig_{CTX} : CTX &\rightarrow \mathcal{P}(S \times \{recv, send\}) \\ (var, sig) &\mapsto sig \end{aligned}$$

The definition of syntactically correct expressions (given in the following sections) is relative to the given context, thus the context encapsulates the expressions' dependencies to the HybridUML specification.

For one technical reason, the context is not defined by the agent which provides the variables and signals: There are (sub-)expressions that are defined within a different context than that of an agent – bound variables  $v_{local} \in V_{local}$  introduced by quantified expressions extend the context for contained expressions, and identifiers embedded into structured data types are defined in the context of the data type.

**Two-Level Syntax.** As a consequence from the dependency of expressions on variables and signals, the HybEL syntax is partitioned into *identifier expressions* and full *hybel expressions*. Identifier expressions define valid identifiers which represent variables or signals from the given context. This includes sub-identifiers  $id_{sdt}.id_{sub}$  of structured data types, as well as indexed identifiers  $id[\langle exp_{idx} \rangle]$ . Full hybel expressions define complete expressions that (can) contain identifier expressions. Note that this is not a two-level definition in the strict sense, but a mutually recursive definition: The index expressions  $exp_{idx}$  within the identifier expressions are full hybel expressions themselves.<sup>2</sup>

The syntax of identifier expressions is given in section 3.2.1, the full hybel identifier syntax is defined in section 3.3.1.

**Semantics.** There are two different ways in which HybEL expressions are interpreted:

*Evaluation Semantics* The expressions from the structural specification part are *evaluated* during the transformation  $\Phi$  from HybridUML specifications into the executable system, when the static structure of the system is created. Effectively, only a subset of the available HybEL expressions is available for the structural specification, because only expressions of type  $t \in \{int, \mathcal{P}(int)\}$  are applied there.

The mapping *eval* provides the associated values. As described in chapter 5, agent instances and therefore also the attached variables are duplicated, and may have different values assigned. This implies that the evaluation of expressions which refer to variables depends on the particular duplicate; thus *eval* is only defined within the transformation  $\Phi$ ,

<sup>2</sup>Technically, this is handled by separate parser runs which omit index expressions at first.

rather than independently. A stand-alone, but less powerful variant  $eval_{\emptyset}$  which omits variables is given in section 3.4. It is sufficient for the syntax definition of hybel expressions in section 3.3.1.

*Transformation Semantics* Expressions from the behavioral specification part as well as property and signal multiplicities from the structural specification part are *transformed* by  $\Phi$  into (parts of) the executable system and evaluated at run-time. For this, the full extent of HybEL is exploited. Obviously, this also depends on the transformation  $\Phi$ , and is discussed in section 5.5 of chapter 5.

In this chapter, *intermediate semantics* are defined for identifier expressions (section 3.2.2) and for full hybel expressions (section 3.3.2). From this, the *evaluation semantics* as well as the *transformation semantics* are defined.

## 3.2 Identifier Expressions

Identifier expressions are subexpressions of HybEL expressions that identify either a variable or a signal within a given context.

For the given HybridUML specification, there is a set  $Id$  of available identifiers. These are associated with the variables and signals from the specification. In addition to the variables  $v \in V$  and signals  $s \in S$ , there are local variables  $v_{local} \in V_{local}$  with  $V \cap V_{local} = \emptyset$  which act as bound variables within expressions. Local variables are always of type *int*. Each variable  $v \in V \cup V_{local}$  or signal  $s \in S$  of the specification has an identifier  $id \in Id$ :

$$\begin{aligned} id_V &: V \rightarrow Id \\ id_{V_{local}} &: V_{local} \rightarrow Id \\ id_{V, V_{local}} &= id_V \cup id_{V_{local}} \\ id_S &: S \rightarrow Id \end{aligned}$$

The sets of identifiers for variables and signals are disjoint:

$$\text{ran } id_V \cap \text{ran } id_{V_{local}} \cap \text{ran } id_S = \emptyset$$

Each identifier of a variable  $v \in V$  and each identifier of a signal  $s \in S$  is unique within the embedding agent or data type, respectively:

$$\begin{aligned} \forall a \in A \bullet \forall v_1, v_2 \in \text{var}_A(a) \bullet id_V(v_1) = id_V(v_2) &\Rightarrow v_1 = v_2 \\ \forall t \in DT \bullet \forall v_1, v_2 \in \text{var}_{DT}(a) \bullet id_V(v_1) = id_V(v_2) &\Rightarrow v_1 = v_2 \\ \forall a \in A \bullet \forall s_1, s_2 \in \text{sig}_A(a) \bullet id_S(s_1) = id_S(s_2) &\Rightarrow s_1 = s_2 \end{aligned}$$

Each local variable has a unique identifier within  $Id$ , i.e. the mapping  $id_{V_{local}}$  is injective.

Available identifiers within a certain context are provided by respective mappings:

$$\begin{aligned} id_{CTX, var} &: CTX \rightarrow \mathcal{P}(Id) \\ c &\mapsto \{id \in Id \mid \exists v \in \text{var}_{CTX}(c) \bullet id_{V, V_{local}}(v) = id\} \\ id_{CTX, sig} &: CTX \rightarrow \mathcal{P}(Id) \\ c &\mapsto \{id \in Id \mid \exists (s, acc) \in \text{sig}_{CTX}(c) \bullet id_V(s) = id\} \end{aligned}$$

$$\begin{aligned}
id_{CTX, sig, recv} &: CTX \rightarrow \mathcal{P}(Id) \\
c &\mapsto \{id \in Id \mid \exists (s, recv) \in sig_{CTX}(c) \bullet id_V(s) = id\} \\
id_{CTX} &: CTX \rightarrow \mathcal{P}(Id) \\
c &\mapsto id_{CTX, var}(c) \cup id_{CTX, sig}(c)
\end{aligned}$$

### 3.2.1 Syntax of Identifier Expressions

The syntax of identifier expressions is given by the mapping

$$syn_{id} : CTX \times CTX \rightarrow \mathcal{P}(\text{seq } \Sigma_{Exp})$$

whereas the alphabet  $\Sigma_{Exp}$  contains all variable and signal identifiers of the HybridUML specification. Additionally, terminal symbols for the separation of structured data type's identifiers, as well as terminal symbols for index expressions of indexed variables or signals are provided:

$$\text{ran } id_V \cup \text{ran } id_{V_{local}} \cup \text{ran } id_S \cup \{., [, ]\} \subset \Sigma_{Exp}$$

The full definition will be given in section 3.3.1.

The syntax of identifier expressions is affected by two contexts: (1) The local context provides the set of variables and signals for which the current identifier must match. (2) A top-level context is needed for embedded index expressions: within structured data types, embedded identifiers can be equipped with index expressions – these have to be evaluated within the context of the outer structured data type identifier, rather than in the local context. Valid identifier expressions are:

- (1) Simple identifiers, identifying a variable or signal:  $id$
- (2) Indexed identifiers, identifying a variable or signal with a given index:  
 $id[17]$
- (3) Signal identifier with written index. This identifies a signal to be received with any index wrt. its multiplicity. When the signal is received at index  $i$ , then  $i$  is stored in the variable that is determined by the index expression. For example:  $id_{sig}[:= id_{int, rw}]$
- (4) Sub-identifier from a simple variable of structured data type, e.g.:  
 $id_{sdt}.id_{sub}$
- (5) Sub-identifier from an indexed variable of structured data type, e.g.:  
 $id_{sdt}[17].id_{sub}$

$$\begin{aligned}
syn_{id}(c, c_{tl}) = & \\
& \{\langle id \rangle \mid id \in id_{CTX}(c)\} \\
\cup & \{\langle id, \rangle \wedge exp_{idx} \wedge \langle \rangle \mid id \in id_{CTX}(c) \wedge exp_{idx} \in syn_{int}(c_{tl})\} \\
\cup & \{\langle id, \rangle \wedge exp_{idx} \wedge \langle \rangle \mid id \in id_{CTX, sig, recv}(c) \wedge exp_{idx} \in syn_{idxass}(c_{tl})\} \\
\cup & \{\langle id, . \rangle \wedge exp_{sub} \mid id \in id_{CTX, var}(c) \wedge \exists v \in var_{CTX}(c) \bullet \\
& (id_{V, V_{local}}(v) = id \wedge exp_{sub} \in syn_{id}((var_{DT}(type_V(v)), \emptyset), c_{tl})\} \\
\cup & \{\langle id, \rangle \wedge exp_{idx} \wedge \langle \rangle \wedge \langle . \rangle \wedge exp_{sub} \mid \\
& id \in id_{CTX, var}(c) \wedge exp_{idx} \in syn_{int}(c_{tl}) \wedge \exists v \in var_{CTX}(c) \bullet \\
& (id_{V, V_{local}}(v) = id \wedge exp_{sub} \in syn_{id}((var_{DT}(type_V(v)), \emptyset), c_{tl})\}
\end{aligned}$$

The syntax of index expressions is defined in section 3.3.1, given by  $syn_{int}$  and  $syn_{idpass}$ , respectively. It is defined as a subset of complete HybEL expressions.

As a shorthand, for equal local context and top-level context, we use

$$\begin{aligned} syn_{id} &: CTX \rightarrow \mathcal{P}(\text{seq } \Sigma_{Exp}) \\ c &\mapsto syn_{id}(c, c) \end{aligned}$$

### 3.2.2 Intermediate Semantics of Identifier Expressions

Expressions  $exp \in \text{seq } \Sigma_{Exp}$  which represent variables or signals are mapped to trees of *identifier items*. Such a tree is the *intermediate semantics* of an identifier expression. In contrast, the final semantics is given in chapter 5, in the context of integrated transformation  $\Phi$  of HybridUML specifications and contained HybEL expressions.

Identifier items consist of a *type* and a *value*:

$$IdItem = IdItemType \times IdItemVal$$

The item type is either the type of the variable or signal, or a special type that identifies a contained index expression. Signal types distinguish signals to be sent and signals to be received.

$$IdItemType = DT \cup \{recvSig, sendSig, indexExp\}$$

The possible item values for items that represent an identifier are either variables  $v \in V \cup V_{local}$  or signals  $s \in S$ , with an attached access policy. In contrast, an item of role *indexExp* represents a full-blown HybEL expression that defines an index wrt. the variable's or signal's multiplicity. Consequently, this expression's value is a complete hybel item tree.<sup>3</sup>

$$IdItemVal = (V \times \{ro, rw\}) \cup (S \times \{recv, send\}) \cup \text{tree}_o \text{ HybelItem}$$

For convenience, mappings to type and access policy are given:

$$\begin{aligned} type_{IdItem} &: IdItem \rightarrow IdItemType \\ (t, v) &\mapsto t \\ acc_{IdItem} &: IdItem \rightarrow \{ro, rw, recv, send, \lambda\} \\ (t, (v, acc)) &\mapsto acc; (v, acc) \in V \times \{ro, rw\} \\ (t, (s, acc)) &\mapsto acc; (s, acc) \in S \times \{recv, send\} \\ item &\mapsto \lambda; \text{ else} \end{aligned}$$

For a tree of identifier items, the type is determined from the contained items, which is the innermost for structured data types  $t \in DT_{struct}$ , otherwise the outermost.

$$\begin{aligned} type_{IdTree} &: \text{tree}_o \text{ IdItem} \rightarrow IdItemType \\ (itm, \langle (itm_1, sub_1), \dots, (itm_n, sub_n) \rangle) &\mapsto type_{IdItem}(itm_n) \\ ; n \geq 1 \wedge type_{IdItem}(itm) \in DT_{struct} \wedge type_{IdItem}(itm_n) \in DT \\ (itm, sub) &\mapsto type_{IdItem}(itm); \text{ else} \end{aligned}$$

<sup>3</sup>Ordered trees are recursively defined as pairs of one node and one sequence of subtrees:  $\text{tree}_o X = X \times \text{seq}(\text{tree}_o X)$

The access policy for a tree of identifier items is recursively composed of the contained items for structured data types  $t \in DT_{struct}$ , otherwise the outermost policy is used.

$$\begin{aligned}
acc_{IdTree} : tree_o IdItem &\rightarrow \{ro, rw, recv, send, \lambda\} \\
(itm, \langle t_1, \dots, t_n \rangle) &\mapsto rw \\
&; n \geq 1 \wedge type_{IdItem}(itm) \in DT_{struct} \\
&\wedge acc_{IdItem}(itm) = rw \wedge acc_{IdTree}(t_n) = rw \\
(itm, \langle t_1, \dots, t_n \rangle) &\mapsto ro \\
&; n \geq 1 \wedge type_{IdItem}(itm) \in DT_{struct} \\
&\wedge (acc_{IdItem}(itm) = ro \vee acc_{IdTree}(t_n) = ro) \\
(itm, sub) &\mapsto acc_{IdItem}(itm); \text{ else}
\end{aligned}$$

The mapping to identifier item trees depends on the given context, as the syntax definition does. Two different contexts are provided here, again as a prerequisite for full identifier expressions: contained index expressions will be mapped corresponding to their embedding identifier expression's *top-level* context, whereas sub-identifiers are related to the context provided by the containing data type. For each pair of contexts, there is a mapping from expressions to corresponding identifier item trees:

$$\begin{aligned}
IT &= \bigcup_{(c_1, c_2) \in CTX \times CTX} (syn_{id}(c_1, c_2) \rightarrow tree_o IdItem) \\
it : CTX \times CTX &\rightarrow IT
\end{aligned}$$

Simple variables  $v \in V$  are represented by a single identifier item that corresponds to the respective type each:

$$\begin{aligned}
it(c, c_{id})(\langle id \rangle) &= ((type_V(v), (v, acc_V(v))), \langle \rangle) \\
&; v \in var_{CTX}(c) \cap V \wedge id_V(v) = id
\end{aligned}$$

Simple signals  $s \in S$  are also represented by single identifiers; the access policy is distinguished here, because signals to be received and signals to be sent are fundamentally different:

$$\begin{aligned}
it(c, c_{id})(\langle id \rangle) &= ((recvSig, (s, recv)), \langle \rangle) \\
&; (s, recv) \in sig_{CTX}(c) \wedge id_S(s) = id \\
it(c, c_{id})(\langle id \rangle) &= ((sendSig, (s, send)), \langle \rangle) \\
&; (s, send) \in sig_{CTX}(c) \wedge id_S(s) = id
\end{aligned}$$

Bound identifiers are implicitly of type *int*, and they are always read-only:

$$\begin{aligned}
it(c, c_{id})(\langle id \rangle) &= ((int, (v, ro)), \langle \rangle) \\
&; v \in var_{CTX}(c) \cap V_{local} \wedge id_{V_{local}}(v) = id
\end{aligned}$$

Simple identifiers with an attached index expression are mapped similarly to simple identifiers without index expression, but from the index expression, a

subtree is created:

$$\begin{aligned}
it(c, c_{tl})(\langle id, l \rangle \hat{\ } exp_{idx} \hat{\ } \langle l \rangle) &= \\
&((type_V(v), (v, acc_V(v))), \langle it(c, c_{tl})(\langle l \rangle \hat{\ } exp_{idx} \hat{\ } \langle l \rangle) \rangle) \\
&; v \in var_{CTX}(c) \cap V \wedge id_V(v) = id \\
it(c, c_{tl})(\langle id, l \rangle \hat{\ } exp_{idx} \hat{\ } \langle l \rangle) &= \\
&((recvSig, (s, recv)), \langle it(c, c_{tl})(\langle l \rangle \hat{\ } exp_{idx} \hat{\ } \langle l \rangle) \rangle) \\
&; (s, recv) \in sig_{CTX}(c) \wedge id_S(s) = id \\
it(c, c_{tl})(\langle id, l \rangle \hat{\ } exp_{idx} \hat{\ } \langle l \rangle) &= \\
&((sendSig, (s, send)), \langle it(c, c_{tl})(\langle l \rangle \hat{\ } exp_{idx} \hat{\ } \langle l \rangle) \rangle) \\
&; (s, send) \in sig_{CTX}(c) \wedge id_S(s) = id
\end{aligned}$$

An identifier expression containing a period defines a sub-identifier that represents a variable from a structured data type. This can be accompanied by an index expression. From the sub-identifier, a subtree is created:

$$\begin{aligned}
it(c, c_{tl})(\langle id, l \rangle \hat{\ } exp_{idx} \hat{\ } \langle l, . \rangle \hat{\ } exp_{suf}) &= \\
&((type_V(v), (v, acc_V(v))), \\
&\langle it(c, c_{tl})(\langle l \rangle \hat{\ } exp_{idx} \hat{\ } \langle l \rangle), it((var_{DT}(type_V(v)), \langle \rangle), c_{tl})(exp_{suf}) \rangle) \\
&; v \in var_{CTX}(c) \cap V \wedge id_V(v) = id \\
it(c, c_{tl})(\langle id, . \rangle \hat{\ } exp_{suf}) &= \\
&((type_V(v), (v, acc_V(v))), \\
&\langle it((var_{DT}(type_V(v)), \langle \rangle), c_{tl})(exp_{suf}) \rangle) \\
&; v \in var_{CTX}(c) \cap V \wedge id_V(v) = id
\end{aligned}$$

An index expression defines a contained hybel item tree, which is interpreted in the attached top-level context:

$$it(c, c_{tl})(\langle l \rangle \hat{\ } exp_{idx} \hat{\ } \langle l \rangle) = ((indexExp, ht(c_{tl})(exp_{idx})), \langle \rangle)$$

The mapping  $ht$  is given in section 3.3.2; it is referenced here because  $ht$  and  $it$  are defined mutually recursively.

As a shorthand, for equal local context and top-level context, we use

$$\begin{aligned}
it : CTX &\rightarrow IT \\
c &\mapsto it(c, c)
\end{aligned}$$

### 3.3 HybEL Expressions

**Alphabet.** The alphabet of  $Exp_{HybEL}$  consists of terminal symbols that denote variable and signal identifiers (as discussed in section 3.2), as well as operations, assignments, and literals. Its full definition is:

$$\begin{aligned}
\Sigma_{Exp} &= \{true, false, \forall, \in, \{, \}, \bullet, (, ), \exists, \neg, +, -, \cdot, /, \hat{\ }, \wedge, \vee, ==, \neq \\
&<, \leq, >, \geq, \dots, \cdot, \cdot, :=, \in, |', [, ], 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\} \\
&\cup L \cup \text{ran } id_V \cup \text{ran } id_{V_{local}} \cup \text{ran } id_S
\end{aligned}$$

### 3.3.1 Syntax of HybEL Expressions

The syntax of HybEL expressions is given by a mapping

$$syn : CTX \rightarrow \mathcal{P}(\text{seq } \Sigma_{Exp})$$

which is partitioned into several sub-mappings. These represent different kinds of expressions, reflecting the applications of expressions within a HybridUML specification as discussed at the beginning of this chapter: (1) Constant expressions, providing values of types  $t \in DT$ . (2) Differential expressions evaluating derivatives of variables of type *anaReal*. (3) Assignment expressions that modify the valuation of a variable. (4) Integer set expressions that calculate finite sets of integers. (5) Special index expressions that assign an index value. (6) Signal raise statements that send signals. (7) Trigger expressions that receive signals and potentially assign variables from the signal's parameters.

$$\begin{aligned} syn(c) = & syn_{const}(c) \cup syn_{diff}(c) \cup syn_{ass}(c) \cup syn_{iset}(c) \cup syn_{idxass}(c) \\ & \cup syn_{sigraise}(c) \cup syn_{trigger}(c) \end{aligned}$$

The set of all HybEL expressions which can occur within a given HybridUML specification is then  $Exp_{HybEL} = \text{ran } syn$ .

**Constant Expressions.** Constant expressions calculate a value of dedicated type, without modifying variables. They are distinguished by the resulting type:

$$\begin{aligned} syn_{const} : CTX & \rightarrow \mathcal{P}(\text{seq } \Sigma_{Exp}) \\ c & \mapsto syn_{bool}(c) \cup syn_{int}(c) \cup syn_{real}(c) \cup \\ & \bigcup_{t \in DT_{enum}} syn_{enum}(c, t) \cup \bigcup_{t \in DT_{struc}} syn_{struc}(c, t) \end{aligned}$$

Boolean expressions are constant expressions which calculate boolean results. They can be defined by:

- (1) Boolean literals.
- (2) Identifiers of boolean variables.
- (3) Unary operation: negation.
- (4) Binary operations on boolean operands.
- (5) Binary operations on numeric operands.
- (6) Binary operations on enumeration-typed operands.
- (7) Quantified boolean expression: A bound expression is evaluated for a finite set of integer values. The conjunction ( $\forall$ ) or disjunction ( $\exists$ ) determines the result. A bound variable is used for access to the integer values. Example:  $\forall i \in \{1..9\} \bullet (x[i] \leq 5)$

$$\begin{aligned} syn_{bool} : CTX & \rightarrow \mathcal{P}(\text{seq } \Sigma_{Exp}) \\ c & \mapsto \{true, false\} \\ & \cup \{exp \in syn_{id}(c) \mid type_{IdTree}(it(c)(exp)) = bool\} \\ & \cup \{\neg \wedge exp \mid exp \in syn_{bool}(c)\} \\ & \cup \{exp_l \wedge \langle \diamond \rangle \wedge exp_r \mid exp_l, exp_r \in syn_{bool}(c) \wedge \diamond \in \{\wedge, \vee, ==, \neq\}\} \\ & \cup \{exp_l \wedge \langle \diamond \rangle \wedge exp_r \mid \end{aligned}$$

$$\begin{aligned}
& \text{exp}_l, \text{exp}_r \in \text{syn}_{int}(c) \cup \text{syn}_{real} \wedge \diamond \in \{<, \leq, >, \geq, ==, \neq\} \\
\cup & \{ \text{exp}_l \hat{\ } \langle \diamond \rangle \hat{\ } \text{exp}_r \mid \exists t \in DT_{enum} \bullet \\
& (\text{exp}_l, \text{exp}_r \in \text{syn}_{enum}(c, t)) \wedge \diamond \in \{==, \neq\} \\
\cup & \{ \langle q, id, \in, \{ \rangle \hat{\ } \text{exp}_{iset} \hat{\ } \langle \}, \bullet, \langle \rangle \hat{\ } \text{exp}_{bound} \hat{\ } \langle \rangle \mid \\
& q \in \{ \forall, \exists \} \wedge id \in \text{ran } id_{V_{local}} \wedge \langle id \rangle \notin \text{syn}_{id}(c) \wedge \text{exp}_{iset} \in \text{syn}_{iset}(c) \\
& \wedge \exists v \in V_{local} \bullet (\text{exp}_{bound} \in \text{syn}_{bool}(\text{var}_{CTX}(c) \cup \{v\}, \text{sig}_{CTX}(c)) \\
& \wedge id = id_{V_{local}}(v)) \}
\end{aligned}$$

Integer expressions are constant expressions of integer type:

- (1) Integer literals.
- (2) Identifiers of integer variables.
- (3) Binary operations on numeric operands.

$$\begin{aligned}
& \text{syn}_{int} : CTX \rightarrow \mathcal{P}(\text{seq } \Sigma_{Exp}) \\
c \mapsto & \{ s \hat{\ } \langle d_1, \dots, d_n \rangle \mid s \in \{ \langle \rangle, \langle - \rangle \} \wedge \forall i \in \{1..n\} \bullet \\
& d_i \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\} \\
\cup & \{ \text{exp} \in \text{syn}_{id}(c) \mid \text{type}_{IdTree}(it(c)(\text{exp})) = int \} \\
\cup & \{ \text{exp}_l \hat{\ } \langle \diamond \rangle \hat{\ } \text{exp}_r \mid \text{exp}_l, \text{exp}_r \in \text{syn}_{int}(c) \wedge \diamond \in \{+, -, \cdot, /, \wedge\} \}
\end{aligned}$$

Real expressions are constant expressions that provide real-valued results:

- (1) Real-valued literals.
- (2) Identifiers of real or analog real variables.
- (3) Binary operations on numeric operands.

$$\begin{aligned}
& \text{syn}_{real} : CTX \rightarrow \mathcal{P}(\text{seq } \Sigma_{Exp}) \\
c \mapsto & \{ s \hat{\ } \langle d_1, \dots, d_k, \cdot, d_{k+1}, d_n \rangle \mid s \in \{ \langle \rangle, \langle - \rangle \} \wedge \forall i \in \{1..n\} \bullet \\
& d_i \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\} \\
\cup & \{ \text{exp} \in \text{syn}_{id}(c) \mid \text{type}_{IdTree}(it(c)(\text{exp})) \in \{real, anaReal\} \} \\
\cup & \{ \text{exp}_l \hat{\ } \langle \diamond \rangle \hat{\ } \text{exp}_r \mid (\text{exp}_l, \text{exp}_r) \in (\text{syn}_{int}(c) \times \text{syn}_{real}(c)) \cup \\
& (\text{syn}_{real}(c) \times \text{syn}_{int}(c)) \cup (\text{syn}_{real}(c) \times \text{syn}_{real}(c)) \\
& \wedge \diamond \in \{+, -, \cdot, /, \wedge\} \}
\end{aligned}$$

Enumeration type expressions are either literals or identifiers of enumeration type. They are distinguished by their concrete data types:

$$\begin{aligned}
& \text{syn}_{enum} : CTX \times DT_{enum} \rightarrow \mathcal{P}(\text{seq } \Sigma_{Exp}) \\
(c, t) \mapsto & L \\
& \cup \{ \text{exp} \in \text{syn}_{id}(c) \mid \text{type}_{IdTree}(it(c)(\text{exp})) = t \}
\end{aligned}$$

Structured data type expressions are either structured literals or identifiers of structured data type. For type compatibility of literals, helper mappings are defined. The mapping  $\text{typeseq}_{DT}$  maps structured data types to their respective list of contained types, which are derived from their properties – each property is first expanded to  $n$  copies according to its multiplicity, then all property copies are concatenated:

$$\text{typeseq}_{DT} : CTX \times DT \rightarrow \text{seq } DT$$

$$\begin{aligned}
(c, t) &\mapsto s_1 \hat{\ } \dots \hat{\ } s_{|varseq_{DT}(t)|} \\
&; \text{kind}_{DT}(t) = \text{struc} \wedge \forall i \in \{1 \dots |varseq_{DT}(t)|\} \bullet \\
&\quad (\text{eval}_{\emptyset}(c)(\text{mult}_V(\text{varseq}_{DT}(t)(i))) \in \mathbb{N} \wedge s_i = \\
&\quad \quad \text{unroll}_{DT \times \mathbb{N}}(\text{type}_V(\text{varseq}_{DT}(t)(i)), \\
&\quad \quad \quad \text{eval}_{\emptyset}(c)(\text{mult}_V(\text{varseq}_{DT}(t)(i)))) \\
(c, t) &\mapsto \langle \rangle; \text{else}
\end{aligned}$$

The expansion of properties according to their multiplicities is defined by the mapping

$$\begin{aligned}
\text{unroll}_{DT \times \mathbb{N}} : DT \times \mathbb{N} &\rightarrow \text{seq } DT \\
(t, n) &\mapsto s_1 \hat{\ } \dots \hat{\ } s_n; \forall i \in \{1..n\} \bullet s_i = \langle t \rangle
\end{aligned}$$

The syntax of structured data type expressions then is defined as the union of all literals and identifiers of variables which are compatible to a given type  $t$ .<sup>4</sup> Note that structured literals can contain arbitrary constant expressions (of appropriate type), not only literals themselves.

- (1) An empty literal fits to structured data types  $t$  which have no properties.
- (2) Literals with exactly one property fit to type  $t$ , iff  $t$  contains exactly one property of the same type. Examples are:  $\{17\}$ ,  $\{id_{sdt}.p_1\}$ ,  $\{\{3.7, x\}\}$
- (3) Literals with several properties fit to type  $t$ , iff  $t$  has the same number and types of properties. Examples are:  $\{3.7, x\}$ ,  $\{\{x, id_{sdt}.p_2\}, 99, id_{sdt}.p_2\}$
- (4) Identifiers of type  $t$  fit to type  $t$ .

$$\begin{aligned}
\text{syn}_{struc} : CTX \times DT_{struc} &\rightarrow \mathcal{P}(\text{seq } \Sigma_{Exp}) \\
(c, t) &\mapsto \{ \langle \{, \} \rangle \mid |typeseq_{DT}(c, t)| = 0 \} \\
&\cup \{ \langle \{ \rangle \hat{\ } exp \hat{\ } \langle \} \rangle \mid |typeseq_{DT}(c, t)| = 1 \wedge \\
&\quad ((typeseq_{DT}(c, t)(1) = \text{bool} \Rightarrow exp \in \text{syn}_{bool}(c)) \\
&\quad \wedge (typeseq_{DT}(c, t)(1) = \text{int} \Rightarrow exp \in \text{syn}_{int}(c)) \\
&\quad \wedge (typeseq_{DT}(c, t)(1) \in \{\text{real}, \text{anaReal}\} \Rightarrow exp \in \text{syn}_{real}(c)) \\
&\quad \wedge (typeseq_{DT}(c, t)(1) \in DT_{enum} \\
&\quad \quad \Rightarrow exp \in \text{syn}_{enum}(c, typeseq_{DT}(c, t)(1))) \\
&\quad \wedge (typeseq_{DT}(c, t)(1) \in DT_{struc} \\
&\quad \quad \Rightarrow exp \in \text{syn}_{struc}(c, typeseq_{DT}(c, t)(1))) \} \\
&\cup \{ \langle \{ \rangle \hat{\ } exp_1 \hat{\ } \langle \cdot \rangle \hat{\ } \dots \hat{\ } \langle \cdot \rangle \hat{\ } exp_n \hat{\ } \langle \} \rangle \mid n = |typeseq_{DT}(c, t)| \\
&\quad \wedge \forall i \in \{1..n\} \bullet \\
&\quad ((typeseq_{DT}(c, t)(i) = \text{bool} \Rightarrow exp_i \in \text{syn}_{bool}(c)) \\
&\quad \wedge (typeseq_{DT}(c, t)(i) = \text{int} \Rightarrow exp_i \in \text{syn}_{int}(c)) \\
&\quad \wedge (typeseq_{DT}(c, t)(i) \in \{\text{real}, \text{anaReal}\} \Rightarrow exp_i \in \text{syn}_{real}(c)) \\
&\quad \wedge (typeseq_{DT}(c, t)(i) \in DT_{enum} \\
&\quad \quad \Rightarrow exp_i \in \text{syn}_{enum}(c, typeseq_{DT}(c, t)(i))) \\
&\quad \wedge (typeseq_{DT}(c, t)(i) \in DT_{struc}
\end{aligned}$$

<sup>4</sup>Note that the semantics  $\text{eval}_{\emptyset}$  (from section 3.4) is used in this definition. The parsing algorithm therefore ensures syntactical correctness of structured data type literals using a separate parser invocation.

$$\begin{aligned} &\Rightarrow \{exp_i \in \text{syn}_{\text{struc}}(c, \text{typeseq}_{DT}(c, t)(i))\} \\ \cup &\{exp \in \text{syn}_{id}(c) \mid \text{type}_{IdTree}(it(c)(exp)) = t\} \end{aligned}$$

**Differential Expressions.** Differential expressions are special constant expressions that refer to derivatives of variables:

- (1) Derivative specification for an identifier of an analog real variable  $v$ . This shall provide the current slope of  $v$  wrt. time, i.e.  $v$  is interpreted as a function  $v : \text{time} \rightarrow \text{anaReal}$ , such that the expression  $v'$  denotes  $\dot{v}(t)$ .
- (2) Operations which themselves contain differential expressions. This allows to combine derivative access and usual variable access.

$$\begin{aligned} \text{syn}_{\text{diff}} &: CTX \rightarrow \mathcal{P}(\text{seq } \Sigma_{Exp}) \\ c &\mapsto \{exp \hat{\ } \langle \rangle \mid exp \in \text{syn}_{id}(c) \wedge \text{type}_{IdTree}(it(c)(exp)) = \text{anaReal}\} \\ &\cup \{exp_l \hat{\ } \langle \diamond \rangle \hat{\ } exp_r \mid (exp_l, exp_r) \in (\text{syn}_{\text{diff}}(c) \times \text{syn}_{\text{diff}}(c)) \cup \\ &\quad (\text{syn}_{\text{diff}}(c) \times \text{syn}_{\text{real}}(c)) \cup (\text{syn}_{\text{real}}(c) \times \text{syn}_{\text{diff}}(c)) \cup \\ &\quad (\text{syn}_{\text{diff}}(c) \times \text{syn}_{\text{int}}(c)) \cup (\text{syn}_{\text{int}}(c) \times \text{syn}_{\text{diff}}(c)) \\ &\quad \wedge \diamond \in \{+, -, \cdot, /, \hat{\ } \}\} \end{aligned}$$

**Assignment Expressions.** Assignments *modify* the valuation of variables. They consist of an identifier of a writable variable of type  $t$  and an expression that calculates a compatible value. Assignments are distinguished by the type of variables that are assigned. Additionally, assignments to and from derivatives of variables are treated separately:

$$\begin{aligned} \text{syn}_{\text{ass}} &: CTX \rightarrow \mathcal{P}(\text{seq } \Sigma_{Exp}) \\ c &\mapsto \text{syn}_{\text{ass}, \text{bool}}(c) \cup \text{syn}_{\text{ass}, \text{int}}(c) \cup \text{syn}_{\text{ass}, \text{real}}(c) \cup \text{syn}_{\text{ass}, \text{anaReal}}(c) \cup \\ &\quad \bigcup_{t \in DT_{\text{enum}}} \text{syn}_{\text{ass}, \text{enum}}(c, t) \cup \bigcup_{t \in DT_{\text{struc}}} \text{syn}_{\text{ass}, \text{struc}}(c, t) \\ &\quad \cup \text{syn}_{\text{ass}, \text{diff}}(c) \end{aligned}$$

Boolean assignments assign a writable boolean variable from a boolean expression. This may be quantified by an *assignment group*, which provides a finite set of integers for which the assignment is done. For example,  $\forall i \in \{0..1, 5..6\} := (id_{\text{bool}}[i] := x[i] \geq 37)$  is a shortcut for the assignments  $id_{\text{bool}}[0] := x[0] \geq 37$ ,  $id_{\text{bool}}[1] := x[1] \geq 37$ ,  $id_{\text{bool}}[5] := x[5] \geq 37$ , and  $id_{\text{bool}}[6] := x[6] \geq 37$ .

$$\begin{aligned} \text{syn}_{\text{ass}, \text{bool}} &: CTX \rightarrow \mathcal{P}(\text{seq } \Sigma_{Exp}) \\ c &\mapsto \{exp_l \hat{\ } \langle := \rangle \hat{\ } exp_r \mid exp_l \in \text{syn}_{id}(c) \\ &\quad \wedge \text{type}_{IdTree}(it(c)(exp_l)) = \text{bool} \wedge \text{acc}_{IdTree}(it(c)(exp_l)) = \text{rw} \\ &\quad \wedge exp_r \in \text{syn}_{\text{bool}}(c)\} \\ \cup &\{ \langle \forall, id, \in, \{ \rangle \hat{\ } exp_{\text{iset}} \hat{\ } \langle \}, :=, \langle \rangle \hat{\ } exp_{\text{bound}} \hat{\ } \langle \rangle \mid \\ &\quad id \in \text{ran } id_{V_{\text{local}}} \wedge \langle id \rangle \notin \text{syn}_{id}(c) \wedge exp_{\text{iset}} \in \text{syn}_{\text{iset}}(c) \\ &\quad \wedge \exists v \in V_{\text{local}} \bullet (exp_{\text{bound}} \in \text{syn}_{\text{ass}, \text{bool}}(\text{var}_{CTX}(c) \cup \{v\}, \text{sig}_{CTX}(c)) \\ &\quad \wedge id = id_{V_{\text{local}}}(v)) \} \end{aligned}$$

Integer assignments assign a writable integer variable from an integer expression, similarly to boolean assignments. Additionally, a *non-deterministic assignment*

from a finite set of integers can be given, that is constrained by a boolean expression. As an example,  $id_{int} := \{i \in \{3..8\} \mid id_{bool}[i] \vee id_{bool}[i + 1]\}$  first determines the subset  $s \subseteq \{3, 4, 5, 6, 7, 8\}$  for which the given boolean expression holds, and then chooses one of the elements for assignment to  $id_{int}$ .

$$\begin{aligned}
& syn_{ass,int} : CTX \rightarrow \mathcal{P}(\text{seq } \Sigma_{Exp}) \\
c & \mapsto \{ exp_l \hat{\ } \langle := \rangle \hat{\ } exp_r \mid exp_l \in syn_{id}(c) \\
& \quad \wedge type_{IdTree}(it(c)(exp_l)) = int \wedge acc_{IdTree}(it(c)(exp_l)) = rw \\
& \quad \wedge exp_r \in syn_{int}(c) \} \\
\cup & \{ \langle \forall, id, \in, \{ \rangle \hat{\ } exp_{iset} \hat{\ } \langle \rangle, :=, \langle \rangle \hat{\ } exp_{bound} \hat{\ } \langle \rangle \mid \\
& \quad id \in \text{ran } id_{V_{local}} \wedge \langle id \rangle \notin syn_{id}(c) \wedge exp_{iset} \in syn_{iset}(c) \\
& \quad \wedge \exists v \in V_{local} \bullet (exp_{bound} \in syn_{ass,int}(var_{CTX}(c) \cup \{v\}, sig_{CTX}(c)) \\
& \quad \wedge id = id_{V_{local}}(v)) \} \\
\cup & \{ exp_{var} \hat{\ } \langle \in, \{, id, \in, \{ \rangle \hat{\ } exp_{iset} \hat{\ } \langle \rangle, \{ \rangle \hat{\ } exp_{bound} \hat{\ } \langle \rangle \mid \\
& \quad exp_{var} \in syn_{id}(c) \wedge type_{IdTree}(it(c)(exp_{var})) = int \\
& \quad \wedge acc_{IdTree}(it(c)(exp_{var})) = rw \wedge id \in \text{ran } id_{V_{local}} \\
& \quad \wedge \langle id \rangle \notin syn_{id}(c) \wedge exp_{iset} \in syn_{iset}(c) \\
& \quad \wedge \exists v \in V_{local} \bullet (exp_{bound} \in syn_{bool}(var_{CTX}(c) \cup \{v\}, sig_{CTX}(c)) \\
& \quad \wedge id = id_{V_{local}}(v)) \}
\end{aligned}$$

Real assignments assign a writable real variable from a real-valued or integer expression. Similarly to boolean assignments, a quantified assignment group can be specified:

$$\begin{aligned}
& syn_{ass,real} : CTX \rightarrow \mathcal{P}(\text{seq } \Sigma_{Exp}) \\
c & \mapsto \{ exp_l \hat{\ } \langle := \rangle \hat{\ } exp_r \mid exp_l \in syn_{id}(c) \wedge \\
& \quad type_{IdTree}(it(c)(exp_l)) = real \\
& \quad \wedge acc_{IdTree}(it(c)(exp_l)) = rw \wedge exp_r \in syn_{int}(c) \cup syn_{real}(c) \} \\
\cup & \{ \langle \forall, id, \in, \{ \rangle \hat{\ } exp_{iset} \hat{\ } \langle \rangle, :=, \langle \rangle \hat{\ } exp_{bound} \hat{\ } \langle \rangle \mid \\
& \quad id \in \text{ran } id_{V_{local}} \wedge \langle id \rangle \notin syn_{id}(c) \wedge exp_{iset} \in syn_{iset}(c) \\
& \quad \wedge \exists v \in V_{local} \bullet (exp_{bound} \in syn_{ass,real}(var_{CTX}(c) \cup \{v\}, sig_{CTX}(c)) \\
& \quad \wedge id = id_{V_{local}}(v)) \}
\end{aligned}$$

Analog real assignments assign a writable analog real variable from a real-valued or integer expression. They are defined in the same way as boolean and real assignments are.

$$\begin{aligned}
& syn_{ass,anaReal} : CTX \rightarrow \mathcal{P}(\text{seq } \Sigma_{Exp}) \\
c & \mapsto \{ exp_l \hat{\ } \langle := \rangle \hat{\ } exp_r \mid exp_l \in syn_{id}(c) \wedge \\
& \quad type_{IdTree}(it(c)(exp_l)) = anaReal \\
& \quad \wedge acc_{IdTree}(it(c)(exp_l)) = rw \wedge exp_r \in syn_{int}(c) \cup syn_{real}(c) \} \\
\cup & \{ \langle \forall, id, \in, \{ \rangle \hat{\ } exp_{iset} \hat{\ } \langle \rangle, :=, \langle \rangle \hat{\ } exp_{bound} \hat{\ } \langle \rangle \mid \\
& \quad id \in \text{ran } id_{V_{local}} \wedge \langle id \rangle \notin syn_{id}(c) \wedge exp_{iset} \in syn_{iset}(c) \\
& \quad \wedge \exists v \in V_{local} \bullet (exp_{bound} \in \\
& \quad \quad syn_{ass,anaReal}(var_{CTX}(c) \cup \{v\}, sig_{CTX}(c)) \\
& \quad \wedge id = id_{V_{local}}(v)) \}
\end{aligned}$$

Enumeration-typed assignments assign a writable variable of enumeration type from an expression of the same type. Again, assignment groups are possible.

$$\begin{aligned}
& syn_{ass,enum} : CTX \times DT_{enum} \rightarrow \mathcal{P}(\text{seq } \Sigma_{Exp}) \\
(c, t) \mapsto & \{ exp_l \hat{\langle} := \rangle \hat{\langle} exp_r \mid exp_l \in syn_{id}(c) \wedge \\
& type_{IdTree}(it(c)(exp_l)) = t \wedge acc_{IdTree}(it(c)(exp_l)) = rw \\
& \wedge exp_r \in syn_{enum}(c, t) \} \\
\cup & \{ \langle \forall, id, \in, \{ \rangle \hat{\langle} exp_{iset} \hat{\langle} \{, :=, \} \rangle \hat{\langle} exp_{bound} \hat{\langle} \{ \rangle \} \mid \\
& id \in \text{ran } id_{V_{local}} \wedge \langle id \rangle \notin syn_{id}(c) \wedge exp_{iset} \in syn_{iset}(c) \\
& \wedge \exists v \in V_{local} \bullet (exp_{bound} \in \\
& syn_{ass,enum}((var_{CTX}(c) \cup \{v\}, sig_{CTX}(c)), t) \\
& \wedge id = id_{V_{local}}(v)) \}
\end{aligned}$$

Structured data type assignments assign a writable variable of structured type from an expression of the same type. Assignment groups are available, too.

$$\begin{aligned}
& syn_{ass,struct} : CTX \times DT_{struct} \rightarrow \mathcal{P}(\text{seq } \Sigma_{Exp}) \\
(c, t) \mapsto & \{ exp_l \hat{\langle} := \rangle \hat{\langle} exp_r \mid exp_l \in syn_{id}(c) \wedge \\
& type_{IdTree}(it(c)(exp_l)) = t \wedge acc_{IdTree}(it(c)(exp_l)) = rw \\
& \wedge exp_r \in syn_{struct}(c, t) \} \\
\cup & \{ \langle \forall, id, \in, \{ \rangle \hat{\langle} exp_{iset} \hat{\langle} \{, :=, \} \rangle \hat{\langle} exp_{bound} \hat{\langle} \{ \rangle \} \mid \\
& id \in \text{ran } id_{V_{local}} \wedge \langle id \rangle \notin syn_{id}(c) \wedge exp_{iset} \in syn_{iset}(c) \\
& \wedge \exists v \in V_{local} \bullet (exp_{bound} \in \\
& syn_{ass,struct}((var_{CTX}(c) \cup \{v\}, sig_{CTX}(c)), t) \\
& \wedge id = id_{V_{local}}(v)) \}
\end{aligned}$$

**Differential Assignments.** Differential assignments assign a writable analog real variable. In contrast to analog real assignments, there must be at least one subexpression which contains derivatives of analog real variables: (1) The right-hand side can contain derivatives of analog real variables  $v_1, \dots, v_n$ , therefore the calculated value is calculated from their evolutions, e.g.  $v := v'_1 + v'_2 + v_3$ . (2) The left-hand side can be the derivative of a writable analog real variable  $v$ , therefore the right-hand side determines the relative change of  $v$ . Examples:  $v' := 2 \cdot v_1$ ,  $v' := 2 \cdot v'_1$ . As with all other assignments, quantification by assignment groups is possible.

$$\begin{aligned}
& syn_{ass,diff} : CTX \rightarrow \mathcal{P}(\text{seq } \Sigma_{Exp}) \\
c \mapsto & \{ exp_l \hat{\langle} := \rangle \hat{\langle} exp_r \mid exp_l \in syn_{id}(c) \\
& \wedge type_{IdTree}(it(c)(exp_l)) = anaReal \\
& \wedge acc_{IdTree}(it(c)(exp_l)) = rw \wedge exp_r \in syn_{diff}(c) \} \\
\cup & \{ exp_l \hat{\langle} \prime, := \rangle \hat{\langle} exp_r \mid exp_l \in syn_{id}(c) \wedge \\
& type_{IdTree}(it(c)(exp_l)) = anaReal \wedge acc_{IdTree}(it(c)(exp_l)) = rw \\
& \wedge exp_r \in syn_{diff}(c) \cup syn_{real}(c) \cup syn_{int}(c) \} \\
\cup & \{ \langle \forall, id, \in, \{ \rangle \hat{\langle} exp_{iset} \hat{\langle} \{, :=, \} \rangle \hat{\langle} exp_{bound} \hat{\langle} \{ \rangle \} \mid \\
& id \in \text{ran } id_{V_{local}} \wedge \langle id \rangle \notin syn_{id}(c) \wedge exp_{iset} \in syn_{iset}(c)
\end{aligned}$$

$$\wedge \exists v \in V_{local} \bullet (exp_{bound} \in syn_{ass,diff}(var_{CTX}(c) \cup \{v\}, sig_{CTX}(c)) \\ \wedge id = id_{V_{local}}(v))\}$$

**Integer Sets.** An integer set expression determines a finite set of integers. It consists of a set of integer ranges, e.g. 1..7, 27..39 identifies the set  $s_{\mathbb{Z}} = \{1..7\} \cup \{27..39\}$ .

$$syn_{iset} : CTX \rightarrow \mathcal{P}(\text{seq } \Sigma_{Exp}) \\ c \mapsto \{exp_l \hat{\ } \langle \cdot \rangle \hat{\ } exp_r \mid exp_l, exp_r \in syn_{int}(c)\} \\ \cup \{exp_l \hat{\ } \langle \cdot \rangle \hat{\ } exp_r \hat{\ } \langle \cdot \rangle \hat{\ } exp_{iset} \mid \\ exp_l, exp_r \in syn_{int}(c) \wedge exp_{iset} \in syn_{iset}(c)\}$$

**Index Assignments.** An index assignment expression identifies a writable variable of type *int* that is supposed to be assigned with an index value (that is always of integer type). It is only applicable as index specification for received signals, see section 3.2.1. Example:  $:= id_{int,rw}$

$$syn_{idxass} : CTX \rightarrow \mathcal{P}(\text{seq } \Sigma_{Exp}) \\ c \mapsto \{\langle := \rangle \hat{\ } exp \mid exp \in syn_{id}(c) \wedge type_{IdTree}(it(c)(exp)) = int \\ \wedge acc_{IdTree}(it(c)(exp)) = rw\}$$

**Signal Raise Statements.** Signal raise statements denote a signal to be raised, optionally equipped with a parameter list, defined by constant expressions.

- (1) A signal without parameters is raised by use of a simple identifier that corresponds to a sendable signal, e.g.:  $id_{sig,send}$
- (2) Alternatively, the empty parameter list can be explicitly given:  $id_{sig,send}()$
- (3) Arbitrary constant expressions can be used for the actual signal parameter. Examples are:  $id_{sig,send}(17)$ ,  $id_{sig,send}(v_1 - v_2)$ .
- (4) Several parameters are given in the usual way:  $id_{sig,send}(p_1, p_2, p_3)$

$$syn_{sigraise} : CTX \rightarrow \mathcal{P}(\text{seq } \Sigma_{Exp}) \\ c \mapsto \{exp \in syn_{id}(c) \mid type_{IdTree}(it(c)(exp)) = sendSig\} \\ \cup \{exp \hat{\ } \langle \cdot \rangle \mid exp \in syn_{id}(c) \wedge type_{IdTree}(it(c)(exp)) = sendSig\} \\ \cup \{exp \hat{\ } \langle \cdot \rangle \hat{\ } pexp \hat{\ } \langle \cdot \rangle \mid exp \in syn_{id}(c) \\ \wedge type_{IdTree}(it(c)(exp)) = sendSig \wedge pexp \in syn_{const}(c)\} \\ \cup \{exp \hat{\ } \langle \cdot \rangle \hat{\ } pexp_1 \hat{\ } \langle \cdot \rangle \hat{\ } \dots \hat{\ } \langle \cdot \rangle \hat{\ } pexp_n \hat{\ } \langle \cdot \rangle \mid \\ exp \in syn_{id}(c) \wedge type_{IdTree}(it(c)(exp)) = sendSig \\ \wedge \forall i \in \{1..n\} \bullet pexp_i \in syn_{const}(c)\}$$

**Trigger Expressions.** Trigger statements define a signal to be received, optionally equipped with a parameter list, which consists of writable variables for parameter reception.

- (1) A trigger without parameters is specified by use of a simple identifier which denotes a receivable signal, e.g.:  $id_{sig,recv}$
- (2) Alternatively, the empty parameter list can be explicitly given:  $id_{sig,recv}()$

- (3) The formal signal parameter is given by an identifier of a writable variable, e.g.:  $id_{sig,recv}(v_{rw})$ .
- (4) Several parameters are given in the usual way:  $id_{sig,recv}(v_{1,rw}, v_{2,rw}, v_{3,rw})$

$$\begin{aligned}
& syn_{trigger} : CTX \rightarrow \mathcal{P}(\text{seq } \Sigma_{Exp}) \\
c \mapsto & \{exp \in syn_{id}(c) \mid type_{IdTree}(it(c)(exp)) = recvSig\} \\
& \cup \{exp \hat{\langle \cdot, \cdot \rangle} \mid exp \in syn_{id}(c) \wedge type_{IdTree}(it(c)(exp)) = recvSig\} \\
& \cup \{exp \hat{\langle \cdot \rangle} \hat{\langle \cdot \rangle} \hat{pexp} \hat{\langle \cdot \rangle} \mid exp \in syn_{id}(c) \\
& \quad \wedge type_{IdTree}(it(c)(exp)) = recvSig \wedge pexp \in syn_{id}(c) \\
& \quad \wedge type_{IdTree}(it(c)(pexp)) \in DT \wedge acc_{IdTree}(it(c)(pexp)) = rw\} \\
& \cup \{exp \hat{\langle \cdot \rangle} \hat{pexp}_1 \hat{\langle \cdot \rangle} \hat{\dots} \hat{\langle \cdot \rangle} \hat{pexp}_n \hat{\langle \cdot \rangle} \mid \\
& \quad exp \in syn_{id}(c) \wedge type_{IdTree}(it(c)(exp)) = recvSig \\
& \quad \wedge \forall i \in \{1..n\} \bullet (pexp_i \in syn_{id}(c) \wedge type_{IdTree}(it(c)(pexp_i)) \in DT \\
& \quad \wedge acc_{IdTree}(it(c)(pexp_i)) = rw)\}
\end{aligned}$$

### 3.3.2 Intermediate Semantics of HybEL Expressions

Expressions  $exp \in Exp_{HybEL}$  are mapped to trees of *hybel items*, which are composed of a *role* and a *value*:

$$HybelItem = HybelItemRole \times HybelItemVal$$

The role of the hybel item is roughly one of operation, assignment, literal, enumeration data type, variable, signal raise statement, trigger, read-only parameter expression, or write-only parameter expression. It also defines the *type* of the item.

$$\begin{aligned}
HybelItemOpRole &= \{(t, op) \mid t \in \{bool, int, real\}\} \cup \{(anaReal, diffOp)\} \\
& \cup \{(bool, q) \mid q \in \{\forall, \exists\}\} \cup \{(intSet, s) \mid s \in \{intSpecs, intRange\}\} \\
HybelItemVarRole &= \{(t, var) \mid t \in DT\} \cup \{(anaReal, derivVar)\} \\
HybelItemLitRole &= \{(t, lit) \mid t \in DT_{enum} \cup \{bool, int, real\} \cup \{sdtanon\}\} \\
HybelItemAssRole &= \{(t, ass) \mid t \in DT\} \cup \{(t, assGroup) \mid t \in DT\} \cup \\
& \quad \{(anaReal, diffAss), (int, intNondetAss), (anaReal, diffAssGroup), \\
& \quad (int, indexAss)\} \\
HybelItemSigRole &= \{(sigtype, d) \mid d \in \{sendSig, recvSig\}\} \\
HybelItemRole &= HybelItemOpRole \cup HybelItemVarRole \\
& \cup HybelItemLitRole \cup HybelItemAssRole \cup HybelItemSigRole
\end{aligned}$$

Depending on the role of the hybel item, an *item value* can be attached. The possible values are operation identifiers, boolean values, integer and real numbers, enumeration literals, as well as trees of *identifier items*. Identifier items represent variables or signals, respectively, which are given by the identifiers from the expression, as discussed in section 3.2.2.

The special value  $\lambda$  denotes the absence of an item value.

$$\begin{aligned}
HybelItemVal &= \{\cdot, /, +, -, \neg, \wedge, \vee, ==, \neq, \leq, <, \geq, >, \hat{\cdot}\} \\
& \cup \mathbb{B} \cup \mathbb{Z} \cup \mathbb{R} \cup L
\end{aligned}$$

$$\begin{aligned} & \cup \text{tree}_o \text{IdItem} \\ & \cup \{\lambda\} \end{aligned}$$

The mapping from expressions to hybel item trees depends on the provided context, similarly to identifier item trees (see section 3.2.2). For each possible context, a mapping from expressions to hybel item trees exists:

$$\begin{aligned} HT &= \bigcup_{c \in CTX} (\text{syn}(c) \rightarrow \text{tree}_o \text{HybelItem}) \\ ht : CTX &\rightarrow HT \end{aligned}$$

Literals of primitive data types are mapped to the respective value:

$$\begin{aligned} ht(c)(\langle \text{lit}_{bool} \rangle) &= (((bool, lit), lit_{bool}), \langle \rangle) \\ &; lit_{bool} \in \{true, false\} \\ ht(c)(\langle d_1, \dots, d_n \rangle) &= (((int, lit), d_1 \dots d_n), \langle \rangle) \\ &; \forall i \in \{1..n\} \bullet d_i \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\} \\ ht(c)(\langle -, d_1, \dots, d_n \rangle) &= (((int, lit), -d_1 \dots d_n), \langle \rangle) \\ &; \forall i \in \{1..n\} \bullet d_i \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\} \\ ht(c)(\langle d_1, \dots, d_{k-1}, ., d_k, \dots, d_n \rangle) &= (((real, lit), d_1 \dots d_{k-1}.d_k \dots d_n), \langle \rangle) \\ &; \forall i \in \{1..n\} \bullet d_i \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\} \\ ht(c)(\langle -, d_1, \dots, d_{k-1}, ., d_k, \dots, d_n \rangle) &= (((real, lit), -d_1 \dots d_{k-1}.d_k \dots d_n), \langle \rangle) \\ &; \forall i \in \{1..n\} \bullet d_i \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\} \end{aligned}$$

Literals of enumeration data types are contained themselves in the hybel item:

$$\begin{aligned} ht(c)(\langle \text{lit}_{enum} \rangle) &= (((dt_L(lit_{enum}), lit), lit_{enum}), \langle \rangle) \\ &; lit_{enum} \in L \end{aligned}$$

Structured data type literals are recursively mapped to a tree representing the contained expressions:

$$\begin{aligned} ht(c)(\langle \{ \} \rangle) &= (((sdt_{anon}, lit), \lambda), \langle \rangle) \\ ht(c)(\langle \{ \} \hat{=} exp \hat{=} \{ \} \rangle) &= (((sdt_{anon}, lit), \lambda), \langle ht(c)(exp) \rangle) \\ &; exp \in \text{syn}_{const}(c) \\ ht(c)(\langle \{ \} \hat{=} exp_1 \hat{=} \langle \rangle \hat{=} \dots \hat{=} \langle \rangle \hat{=} exp_n \hat{=} \{ \} \rangle) &= \\ &(((sdt_{anon}, lit), \lambda), \langle ht(c)(exp_1), \dots, ht(c)(exp_n) \rangle) \\ &; \forall i \in \{1..n\} \bullet exp_i \in \text{syn}_{const}(c) \end{aligned}$$

Variable identifiers are mapped to the identifier item tree that represents the respective variable:

$$\begin{aligned} ht(c)(exp) &= (((type_{IdTree}(it(c)(exp)), var), it(c)(exp)), \langle \rangle) \\ &; exp \in \text{syn}_{id}(c) \wedge type_{IdTree}(it(c)(exp)) \in DT \end{aligned}$$

Signal identifiers are mapped to the identifier item tree that represents the respective signal, potentially including parameters:

$$ht(c)(exp) = (((sigtype, type_{IdTree}(it(c)(exp))), it(c)(exp)), \langle \rangle)$$

$$\begin{aligned}
& ; exp \in syn_{id}(c) \wedge type_{IdTree}(it(c)(exp)) \in \{recvSig, sendSig\} \\
ht(c)(exp \hat{\ } \langle \rangle \hat{\ } pexp \hat{\ } \langle \rangle) & \\
= (((sigtype, type_{IdTree}(it(c)(exp))), it(c)(exp)), \langle ht(c)(pexp) \rangle) & \\
; exp \in syn_{id}(c) \wedge type_{IdTree}(it(c)(exp)) \in \{recvSig, sendSig\} & \\
\wedge pexp \in syn_{const}(c) \cup syn_{id}(c) & \\
ht(c)(exp \hat{\ } \langle \rangle \hat{\ } pexp_1 \hat{\ } \langle \rangle \hat{\ } \dots \hat{\ } \langle \rangle \hat{\ } pexp_n \hat{\ } \langle \rangle) = & \\
(((sigtype, type_{IdTree}(it(c)(exp))), it(c)(exp)), & \\
\langle ht(c)(pexp_1), \dots, ht(c)(pexp_n) \rangle) & \\
; exp \in syn_{id}(c) \wedge type_{IdTree}(it(c)(exp)) \in \{recvSig, sendSig\} & \\
\wedge \forall i \in \{1..n\} \bullet pexp_i \in syn_{const}(c) \cup syn_{id}(c) &
\end{aligned}$$

Derivatives are mapped to the identifier item tree that represents the derived variable, which is always of type *anaReal*.

$$\begin{aligned}
ht(c)(exp \hat{\ } \langle \rangle) &= (((anaReal, derivVar), it(c)(exp)), \langle \rangle) \\
; exp \in syn_{id}(c) \wedge type_{IdTree}(it(c)(exp)) &= anaReal
\end{aligned}$$

Simple operations with boolean result are recursively mapped to a tree that represents the operation, containing a subtree for each operand.<sup>5</sup>

$$\begin{aligned}
ht(c)(\langle \neg \rangle \hat{\ } exp) &= (((bool, op), \neg), \langle ht(c)(exp) \rangle) \\
ht(c)(exp_1 \hat{\ } \langle \diamond \rangle \hat{\ } exp_2) &= (((bool, op), \diamond), \langle ht(c)(exp_1), ht(c)(exp_2) \rangle) \\
; \diamond \in \{==, \wedge, \vee, \neq, <, \leq, >, \geq\} &
\end{aligned}$$

Simple operations with numeric result are recursively mapped to a tree that represents the operation, containing a subtree for each operand. The role is determined by the roles of the subtrees.<sup>5</sup>

$$\begin{aligned}
ht(c)(exp_l \hat{\ } \langle \diamond \rangle \hat{\ } exp_r) &= \\
(((int, op), \diamond), \langle ht(c)(exp_l), ht(c)(exp_r) \rangle) & \\
; exp_l \hat{\ } \langle \diamond \rangle \hat{\ } exp_r \in syn_{int}(c) & \\
\wedge \diamond \in \{+, -, \cdot, /, \hat{\ } \} & \\
ht(c)(exp_l \hat{\ } \langle \diamond \rangle \hat{\ } exp_r) &= \\
(((real, op), \diamond), \langle ht(c)(exp_l), ht(c)(exp_r) \rangle) & \\
; exp_l \hat{\ } \langle \diamond \rangle \hat{\ } exp_r \in syn_{real}(c) & \\
\wedge \diamond \in \{+, -, \cdot, /, \hat{\ } \} & \\
ht(c)(exp_l \hat{\ } \langle \diamond \rangle \hat{\ } exp_r) &= \\
(((anaReal, diffOp), \diamond), \langle ht(c)(exp_l), ht(c)(exp_r) \rangle) & \\
; exp_l \hat{\ } \langle \diamond \rangle \hat{\ } exp_r \in syn_{diff}(c) & \\
\wedge \diamond \in \{+, -, \cdot, /, \hat{\ } \} &
\end{aligned}$$

Quantified expressions are mapped to an item that holds an identifier tree which represents the bound variable that is introduced by the quantification. There are two children: (1) An expression defining a finite set of integers – these are

<sup>5</sup>The usual operator precedence is guaranteed by the parsing algorithm. It is omitted in this presentation.

the values that the bound variable adopts. (2) An expression that shall be evaluated for each value of the bound variable.

$$\begin{aligned}
ht(c)(\langle \forall, id, \in, \{ \rangle \wedge exp_{iset} \wedge \langle \}, \bullet, \langle \rangle \wedge exp_{bound} \wedge \langle \rangle) &= \\
&(((bool, \forall), it(c)(\langle id \rangle)), \\
&\langle ht(c)(exp_{iset}), ht((var_{CTX}(c) \cup \{v\}, sig_{CTX}(c))(exp_{bound})) \rangle \\
&; v \in V_{local} \wedge id_{V_{local}}(v) = id \\
ht(c)(\langle \exists, id, \in, \{ \rangle \wedge exp_{iset} \wedge \langle \}, \bullet, \langle \rangle \wedge exp_{bound} \wedge \langle \rangle) &= \\
&(((bool, \exists), it(c)(\langle id \rangle)), \\
&\langle ht(c)(exp_{iset}), ht((var_{CTX}(c) \cup \{v\}, sig_{CTX}(c))(exp_{bound})) \rangle \\
&; v \in V_{local} \wedge id_{V_{local}}(v) = id
\end{aligned}$$

An integer set specification is represented by an item that contains a list of integer range expressions:

$$\begin{aligned}
ht(c)(exp) &= (((intSet, intSpecs), \lambda), \langle sexp \rangle) \\
&; \exists exp_l, exp_r \in syn_{int}(c) \bullet \\
&\quad (exp = exp_l \wedge \langle \cdot \rangle \wedge exp_r \\
&\quad \wedge sexp = (((intSet, intRange), \lambda), \langle ht(c)(exp_l), ht(c)(exp_r) \rangle)) \\
ht(c)(exp_1 \wedge \langle \cdot \rangle \wedge \dots \wedge \langle \cdot \rangle \wedge exp_n) &= \\
&(((intSet, intSpecs), \lambda), \langle sexp_1, \dots, sexp_n \rangle) \\
&; \forall i \in \{1..n\} \bullet \exists exp_l, exp_r \in syn_{int}(c) \bullet \\
&\quad (exp_i = exp_l \wedge \langle \cdot \rangle \wedge exp_r \\
&\quad \wedge sexp_i = (((intSet, intRange), \lambda), \langle ht(c)(exp_l), ht(c)(exp_r) \rangle))
\end{aligned}$$

Simple assignments contain two subexpressions: (1) the left-hand side, consisting of a variable identifier, and (2) the right-hand side, representing a constant expression of appropriate type.

$$\begin{aligned}
ht(c)(exp_l \wedge \langle := \rangle \wedge exp_r) &= (((bool, ass), \lambda), \langle ht(c)(exp_l), ht(c)(exp_r) \rangle) \\
&; exp_l \wedge \langle := \rangle \wedge exp_r \in syn_{ass, bool}(c) \\
ht(c)(exp_l \wedge \langle := \rangle \wedge exp_r) &= (((int, ass), \lambda), \langle ht(c)(exp_l), ht(c)(exp_r) \rangle) \\
&; exp_l \wedge \langle := \rangle \wedge exp_r \in syn_{ass, int}(c) \\
ht(c)(exp_l \wedge \langle := \rangle \wedge exp_r) &= (((real, ass), \lambda), \langle ht(c)(exp_l), ht(c)(exp_r) \rangle) \\
&; exp_l \wedge \langle := \rangle \wedge exp_r \in syn_{ass, real}(c) \\
ht(c)(exp_l \wedge \langle := \rangle \wedge exp_r) &= (((anaReal, ass), \lambda), \langle ht(c)(exp_l), ht(c)(exp_r) \rangle) \\
&; exp_l \wedge \langle := \rangle \wedge exp_r \in syn_{ass, anaReal}(c) \\
ht(c)(exp_l \wedge \langle := \rangle \wedge exp_r) &= (((t, ass), \lambda), \langle ht(c)(exp_l), ht(c)(exp_r) \rangle) \\
&; exp_l \wedge \langle := \rangle \wedge exp_r \in syn_{ass, enum}(c, t) \\
ht(c)(exp_l \wedge \langle := \rangle \wedge exp_r) &= (((t, ass), \lambda), \langle ht(c)(exp_l), ht(c)(exp_r) \rangle) \\
&; exp_l \wedge \langle := \rangle \wedge exp_r \in syn_{ass, struc}(c, t)
\end{aligned}$$

Differential assignments are either assignments to the derivative of a variable, or assignments which are calculated from derivatives of variables:

$$ht(c)(exp_l \wedge \langle := \rangle \wedge exp_r) =$$

$$\begin{aligned} &(((anaReal, diffAss), \lambda), (ht(c)(exp_l), ht(c)(exp_r))) \\ &; exp_l \hat{=} exp_r \in syn_{ass, diff}(c) \end{aligned}$$

A non-deterministic integer assignment is composed of four parts: (1) the variable that is assigned, (2) a bound variable, (3) an integer set specification, and (4) a bound expression:

$$\begin{aligned} &ht(c)(exp_{var} \hat{=} \langle \in, \iota, id, \in, \iota \rangle \hat{=} exp_{iset} \hat{=} \langle \iota, \iota \rangle \hat{=} exp_{bound} \hat{=} \langle \iota \rangle) = \\ &(((int, intNondetAss), it(c)(\langle id \rangle)), \\ &\langle ht(c)(exp_{var}), ht(c)(exp_{iset}), \\ &ht((var_{CTX}(c) \cup \{v\}, sig_{CTX}(c))(exp_{bound})) \rangle) \\ &; v \in V_{local} \wedge id_{V_{local}}(v) = id \end{aligned}$$

A group of assignments consists of a quantified assignment with corresponding type:

$$\begin{aligned} &ht(c)(\langle \forall, id, \in, \iota \rangle \hat{=} exp_{iset} \hat{=} \langle \iota, \iota \rangle \hat{=} exp_{bound} \hat{=} \langle \iota \rangle) = \\ &(((bool, assGroup), it(c)(\langle id \rangle)), \\ &\langle ht(c)(exp_{iset}), ht((var_{CTX}(c) \cup \{v\}, sig_{CTX}(c))(exp_{bound})) \rangle) \\ &; v \in V_{local} \wedge id_{V_{local}}(v) = id \\ &\wedge exp_{bound} \in syn_{ass, bool}((var_{CTX}(c) \cup \{v\}, sig_{CTX}(c))) \\ &ht(c)(\langle \forall, id, \in, \iota \rangle \hat{=} exp_{iset} \hat{=} \langle \iota, \iota \rangle \hat{=} exp_{bound} \hat{=} \langle \iota \rangle) = \\ &(((int, assGroup), it(c)(\langle id \rangle)), \\ &\langle ht(c)(exp_{iset}), ht((var_{CTX}(c) \cup \{v\}, sig_{CTX}(c))(exp_{bound})) \rangle) \\ &; v \in V_{local} \wedge id_{V_{local}}(v) = id \\ &\wedge exp_{bound} \in syn_{ass, int}((var_{CTX}(c) \cup \{v\}, sig_{CTX}(c))) \\ &ht(c)(\langle \forall, id, \in, \iota \rangle \hat{=} exp_{iset} \hat{=} \langle \iota, \iota \rangle \hat{=} exp_{bound} \hat{=} \langle \iota \rangle) = \\ &(((real, assGroup), it(c)(\langle id \rangle)), \\ &\langle ht(c)(exp_{iset}), ht((var_{CTX}(c) \cup \{v\}, sig_{CTX}(c))(exp_{bound})) \rangle) \\ &; v \in V_{local} \wedge id_{V_{local}}(v) = id \\ &\wedge exp_{bound} \in syn_{ass, real}((var_{CTX}(c) \cup \{v\}, sig_{CTX}(c))) \\ &ht(c)(\langle \forall, id, \in, \iota \rangle \hat{=} exp_{iset} \hat{=} \langle \iota, \iota \rangle \hat{=} exp_{bound} \hat{=} \langle \iota \rangle) = \\ &(((anaReal, assGroup), it(c)(\langle id \rangle)), \\ &\langle ht(c)(exp_{iset}), ht((var_{CTX}(c) \cup \{v\}, sig_{CTX}(c))(exp_{bound})) \rangle) \\ &; v \in V_{local} \wedge id_{V_{local}}(v) = id \\ &\wedge exp_{bound} \in syn_{ass, anaReal}((var_{CTX}(c) \cup \{v\}, sig_{CTX}(c))) \\ &ht(c)(\langle \forall, id, \in, \iota \rangle \hat{=} exp_{iset} \hat{=} \langle \iota, \iota \rangle \hat{=} exp_{bound} \hat{=} \langle \iota \rangle) = \\ &(((t, assGroup), it(c)(\langle id \rangle)), \\ &\langle ht(c)(exp_{iset}), ht((var_{CTX}(c) \cup \{v\}, sig_{CTX}(c))(exp_{bound})) \rangle) \\ &; v \in V_{local} \wedge id_{V_{local}}(v) = id \wedge t \in DT_{enum} \\ &\wedge exp_{bound} \in syn_{ass, enum}((var_{CTX}(c) \cup \{v\}, sig_{CTX}(c)), t) \\ &ht(c)(\langle \forall, id, \in, \iota \rangle \hat{=} exp_{iset} \hat{=} \langle \iota, \iota \rangle \hat{=} exp_{bound} \hat{=} \langle \iota \rangle) = \\ &(((t, assGroup), it(c)(\langle id \rangle)), \\ &\langle ht(c)(exp_{iset}), ht((var_{CTX}(c) \cup \{v\}, sig_{CTX}(c))(exp_{bound})) \rangle) \end{aligned}$$

$$\begin{aligned} & ; v \in V_{local} \wedge id_{V_{local}}(v) = id \wedge t \in DT_{struct} \\ & \wedge exp_{bound} \in syn_{ass, struct}((var_{CTX}(c) \cup \{v\}, sig_{CTX}(c)), t) \end{aligned}$$

A group of differential assignments consists of a quantified differential assignment:

$$\begin{aligned} & ht(c)(\langle \forall, id, \in, \{ \rangle \wedge exp_{iset} \wedge \langle \}, :=, \langle \rangle \wedge exp_{bound} \wedge \langle \rangle \rangle) = \\ & (((anaReal, diffAssGroup), it(c)(\langle id \rangle)), \\ & \langle ht(c)(exp_{iset}), ht((var_{CTX}(c) \cup \{v\}, sig_{CTX}(c)))(exp_{bound}) \rangle) \\ & ; v \in V_{local} \wedge id_{V_{local}}(v) = id \wedge \\ & \wedge exp_{bound} \in syn_{ass, diff}((var_{CTX}(c) \cup \{v\}, sig_{CTX}(c))) \end{aligned}$$

An index assignment contains a variable of integer type, such that an index value can be assigned to it:

$$ht(c)(\langle := \rangle \wedge exp_{intvar}) = (((int, indexAss), \lambda), \langle ht(c)(exp_{intvar}) \rangle)$$

### 3.4 Skeleton Evaluation Semantics of HyBEL Expressions

In this section, a skeleton semantics for the evaluation of HyBEL expressions is given. Its main purpose for this chapter is to define the value for multiplicity expressions of sub-properties of structured data types, as it is used in section 3.3.1. The definition of the full evaluation semantics is provided in section 5.2.

The *skeleton evaluation semantics* of hybel expressions is given by

$$\begin{aligned} VAL_{eval} &= \mathbb{B} \cup \mathbb{Z} \cup \mathbb{R} \cup L \cup seq \ VAL_{eval} \cup \mathcal{P}(\mathbb{Z}) \\ MAP_{VAL_{eval}} &= \bigcup_{c \in CTX} (syn(c) \rightarrow VAL_{eval} \cup \{\lambda\}) \\ eval_{\emptyset} : CTX &\rightarrow MAP_{VAL_{eval}} \\ eval_{\emptyset}(c)(exp) &= eval_{ht, \emptyset}(ht(c)(exp)) \end{aligned}$$

Each hybel item tree is mapped to either a value, or to the special value  $\lambda$  that denotes the absence of a value:<sup>6</sup>

$$eval_{ht, \emptyset} : tree_o \ HybelItem \rightarrow VAL_{eval} \cup \{\lambda\}$$

Literal values *are* the values:

$$\begin{aligned} & (((t, lit), l), sub) \mapsto l; t \in DT \\ & (((sdt_{anon}, lit), v), \langle t_1, \dots, t_n \rangle) \mapsto \langle eval_{ht, \emptyset}(t_1), \dots, eval_{ht, \emptyset}(t_n) \rangle \end{aligned}$$

Operations are evaluated:

$$\begin{aligned} & (((t, op), \diamond), \langle t_1, t_2 \rangle) \mapsto eval_{ht, \emptyset}(t_1) \diamond eval_{ht, \emptyset}(t_2) \\ & ; t_1, t_2 \in \{int, real, anaReal\} \wedge \diamond \in \{+, -, \cdot, /, \wedge, <, \leq, \geq, >\} \\ & \vee t_1 = t_2 \wedge \diamond \in \{==, \neq\} \\ & \vee t_1, t_2 \in \{bool\} \wedge \diamond \in \{\wedge, \vee\} \\ & (((t, op), \neg), \langle t_1 \rangle) \mapsto \neg eval_{ht, \emptyset}(t_1) \end{aligned}$$

<sup>6</sup>Note that values  $v \in \mathcal{P}(\mathbb{Z})$  do not occur for  $eval_{ht, \emptyset}$ , but will for the full evaluation semantics in section 5.2.

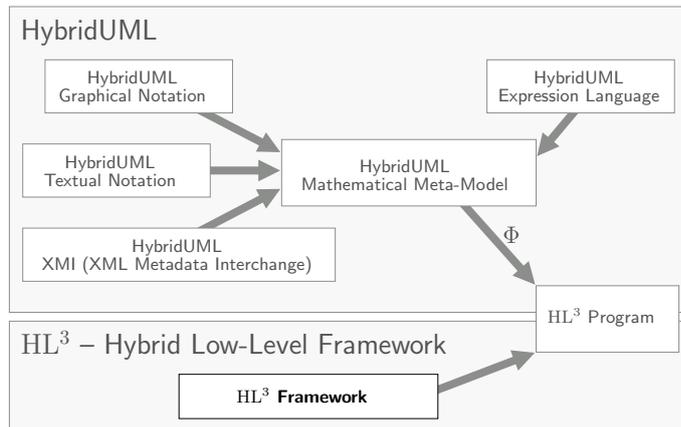
No other expressions are successfully evaluated:

$t \mapsto \lambda$ ; else



## Chapter 4

# HL<sup>3</sup> – Hybrid Low-Level Framework



In this chapter, the HL<sup>3</sup> Low-Level Framework is defined, a compilation target for hybrid systems specification formalisms. HL<sup>3</sup> restricts the variety of executable programs to a fixed structure – the HL<sup>3</sup> design pattern – in a way that is supposed to be adequate for arbitrary formalisms that provide the modeling of hybrid systems. Any transformation  $\Phi$  from such a specification formalism into HL<sup>3</sup> must preserve these restrictions.

Further, HL<sup>3</sup> provides fixed components that can be used by transformations  $\Phi$  to construct HL<sup>3</sup> models from corresponding high-level models. The available components are called the HL<sup>3</sup> runtime environment.

A formal operational semantics is given for the execution of HL<sup>3</sup> models. Therefore, HL<sup>3</sup> models are defined as a mixture of explicit program code and abstractions to mathematical representations.

The specific program code of a specific model, and parts of the abstractly defined behavior, depend (1) on the particular specification formalism, as well as (2) on the specific high-level model to be transformed. These parts have to be added by the corresponding transformation  $\Phi$ .

The *Hybrid Low-Level Framework* HL<sup>3</sup> is a generic compilation target for hybrid high-level formalisms. It is designed to support the transformation of

high-level specifications into executable code, thereby assigning a formal semantics to the generated HL<sup>3</sup> *model*. The HL<sup>3</sup> model (also called HL<sup>3</sup> program) is suitable for hard real-time execution, to be used either for developing embedded applications or for their automated test in hardware-in-the-loop configurations.

In the following, we focus on the *execution* of HL<sup>3</sup> models – we define an operational semantics for the execution of HL<sup>3</sup> models.

**Operational Semantics of HL<sup>3</sup>.** The operational semantics of HL<sup>3</sup> models is given by a state transition system (STS). The transition system  $sts = (S, s_0, T)$  defines the semantics of a specific HL<sup>3</sup> model. It consists of a set  $S$  of states, an initial state  $s_0 \in S$ , and a transition relation  $T \subseteq S \times S$ .

The utilization of STS instead of labeled transition systems (LTS) emphasizes the fact that HL<sup>3</sup> just operates on states. Events – which would occur as labels of an LTS – are considered as a higher level concept, to be “implemented” in HL<sup>3</sup> as state changes of dedicated variables.

For the operational semantics of HL<sup>3</sup> the state space  $S$  is built by a cross product

$$S = CONST \times VAR$$

The sub-vector  $c \in CONST$  of states  $(c, v) \in S$  represents the HL<sup>3</sup> model which results from the transformation of a high-level model into the HL<sup>3</sup> framework. It is the same for every state  $s \in S$ , thus it remains constant under any transition of  $sts$ . Nevertheless, it is useful to consider  $c$  as part of the state space, since application conditions for transitions of the STS may depend on its values.

The dynamic part of the states of  $sts$  is then given by  $v \in VAR$ . This is the “conventional” part, encoding control state, valuations, etc. Both  $CONST$  and  $VAR$  are presented in detail in sections 4.2 and 4.3, respectively.

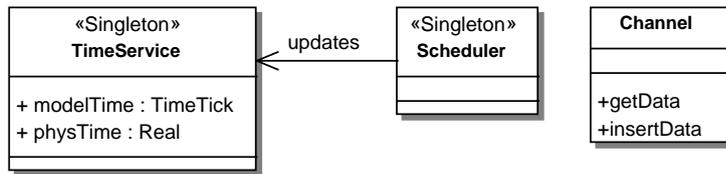
The details of  $sts$  are provided in the subsequent sections. They are structured as follows: We start with an informal overview of the HL<sup>3</sup> framework in section 4.1, which particularly identifies the entities of HL<sup>3</sup> models. Sections 4.2 and 4.3 then formally define the state space  $S$  of the operational semantics of HL<sup>3</sup> models. The transitions  $t \in T$  of the operational semantics, that define the system’s behavior, are defined in sections 4.4ff – section 4.4 provides the system’s overall behavior, i.e. the scheduling of active entities, and sections 4.5 and 4.6 contain the behavior of active entities themselves.

## 4.1 HL<sup>3</sup> Overview

HL<sup>3</sup> provides a re-usable hard real-time processing infrastructure – the *runtime environment* – and a *design pattern* for the formalism- and specification-dependent components to be executed within the runtime environment. An HL<sup>3</sup> model then consists of a set of passive objects that constitute the runtime environment, and a set of active objects that implement the design pattern.

### 4.1.1 Runtime Environment

The HL<sup>3</sup> *runtime environment* provides pre-defined entities that are available for the instantiation of an HL<sup>3</sup> model.

Figure 4.1: Runtime Environment of the HL<sup>3</sup> framework.

**TimeService.** At the heart of a real-time system, there is a notion of *time*. The runtime environment provides a *TimeService* which relates the *physical time* to a *model time*. Both are synchronized during the execution of an HL<sup>3</sup> model, such that the model execution is based on the model time, which is a discretized view on the physical time.

Physical time is a *global time*, since we assume that HL<sup>3</sup> models are executed locally, i.e. on a cluster connected by high-speed local area networks. Therefore, relativistic effects between cluster nodes are neglected. Model time can be obtained by all objects of the model as a pair  $t_0.t_1$ , with component  $t_0$  representing the discretized physical time. As long as  $t_0$  is kept constant, calculations take place in zero time, from the model's perspective. The second component  $t_1$  is then used to distinguish *causally* related calculations which occur during the same time tick  $t_0$ . Consequently,  $t_1$  is always reset when  $t_0$  is incremented.

**Channel.** In order to model a consistent view on global model data which can be transparently distributed over the (hardware) system, the data structure *Channel* is available. Channels can store several copies of values for different recipients, such that specific values are published at different model times. Typically, on read access, the newest value which is addressed to the respective recipient is obtained. That is, among all data items contained in the channel for which the recipient is within their scope, the most recent entry associated with a visibility time that is less or equal to the current model time, is chosen.

Different requirements on HL<sup>3</sup> executions can be satisfied by use of channels: (1) Racing conditions between calculations that are executed in parallel can be avoided, by publishing calculation results in the model future. Since read access always takes place in model presence, there are different values for the same channel, and they do not conflict. (2) Simultaneous calculations from the model view can be executed sequentially, such that the results of the former calculations are published after the last one is finished. (3) For the execution on (cluster) hardware architectures, the distribution of data takes physical time. By choosing an appropriate delay for data publication, a consistent view of data at all (cluster) nodes can be achieved: Every write access to a channel leads to immediate distribution of the data within the whole cluster. As long as the distribution is completed before the data becomes visible, all cluster nodes will have a consistent view on this data.

Further, specific causalities wrt. time, as required by specific high-level formalisms, can be modeled. For example, different publication times can be used for formalisms where changes shall become immediately visible within the local context of an executing entity, but are published later to external ones.

**Scheduler.** The central instance of the model execution is the *Scheduler*, since it defines the *cooperation* of the active objects from the design pattern. The scheduler defines an execution loop with specific execution phases, it defines the sequence of object executions and their distribution to CPUs (or cluster nodes). For periodic executions, a system period is determined at compile time, such that the execution loop is synchronized with physical time.

Implementations of the HL<sup>3</sup> scheduler require a set of reserved CPUs on which the active objects can be executed without interruptions from an underlying operating system.

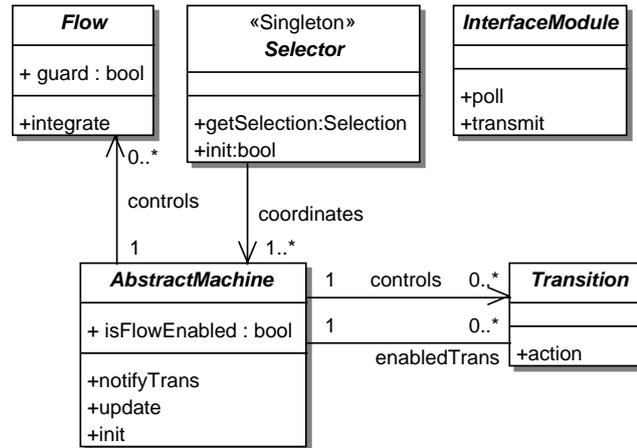


Figure 4.2: Design Pattern of the HL<sup>3</sup> framework.

### 4.1.2 Design Pattern

For hybrid high-level formalisms there are two fundamentally different kinds of behavioral steps which occur within an execution of a model: (1) *Flows* define continuous evolutions of values wrt. time. For example, they can represent algebraic or differential equations from a high-level formalism. (2) Discrete steps represent instantaneous calculations and have no time duration – they are atomic state transformers. Discrete steps are usually called *transitions*. Transitions within HL<sup>3</sup> are different from e.g. statechart transitions, in that they represent discrete steps in an *abstract* way. There are neither source or target states for transitions, nor are transitions equipped with signals/events or guard conditions. These are specific concepts of particular high-level formalisms to be implemented in HL<sup>3</sup>.<sup>1</sup>

The flows and transitions of a high-level formalism are *coordinated* by control behavior, which can be sequential and/or parallel. For the representation of control behavior from the high-level formalism, the HL<sup>3</sup> framework provides sequential control components called *abstract machines*. The abstract machines of an HL<sup>3</sup> model are always executed concurrently, such that parallel behavior can be modeled, too.

<sup>1</sup>Further note that HL<sup>3</sup> transitions are part of the state space  $S$  and are distinct from the transitions  $t \in T$  of our HL<sup>3</sup> operational semantics.

Flows, transitions, and abstract machines are active objects within the HL<sup>3</sup> framework. We constrain them such that each flow and each transition is controlled by a specific abstract machine. Further, within an HL<sup>3</sup> model a central instance is required which coordinates the set of concurrent abstract machines – the *selector*. Its name is motivated by the fact that its purpose is to make a selection from a set of possible steps, in certain situations. It enforces global behavioral constraints on the concurrent abstract machines, such as the synchronous execution of transitions. Since these constraints do not depend on the concrete high-level specifications, the selector has to be defined only per high-level formalism. Nevertheless, it can be useful to apply different selectors for the same formalism: (1) For application development, a selector will usually resolve nondeterministic transition selection – which may be allowed according to the high-level formalism – to deterministic execution sequences. (2) In contrast to this, a simulation or testing system will require a selector which is capable of producing all transition schedules possible according to the high-level formalism.

For the usage of HL<sup>3</sup>, it is required that the intended behavior of the applied high-level formalism can be decomposed into these components, i.e. into flows, transitions, abstract machines, and a selector. We expect that this is the case for all reasonable hybrid high-level formalisms.

Additionally, since hybrid specifications are mostly useful when they are connected to a physical environment, a hardware abstraction layer is given which hides driver-specific details and the location of hardware interfaces – so-called *interface modules* encapsulate external interfaces.

The flows, transitions, abstract machines, interface modules, and the selector constitute the entire set of active objects of an HL<sup>3</sup> model. Within the HL<sup>3</sup> framework their *responsibilities* are defined, but their behavior *is not*. This has to be defined by use of an instantiation rule which depends on the applied high-level formalism. According to this rule, models of the high-level formalism are then transformed into HL<sup>3</sup> models and define specific active HL<sup>3</sup> objects. Therefore, the definition of their responsibilities is called *design pattern*.

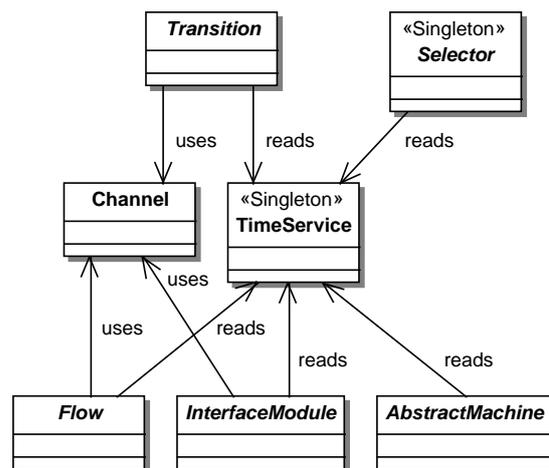


Figure 4.3: Channels and the time service are accessible to the active objects of the design pattern.

The responsibilities of the active objects are:

**Flow** Since the aim of the HL<sup>3</sup> framework is to provide executable models, flows are *discretized*. HL<sup>3</sup> flows require integration functions which can be called with regular frequency, such that each flow may be activated with this specific frequency. Its `guard` attribute then determines whether the flow is actually activated or not. The single responsibility of a flow is:

*integrate()* A flow provides an operation `integrate()` which calculates a single discrete step for the approximation of its continuous specification. The calculation's pre-state is retrieved from channels, and the calculation's results (i.e. the post-state) is written (back) to channels, for publication in the future.

**Transition** The effect of a transition is an inherently discrete calculation:

*action()* The transition's action is the effect of the operation `action()`. It is implemented as a function reading channel data pre-state and setting post state via channels, in the same fashion as flow steps are. In contrast, it is executed exactly once when selected. Further, the publication time depends on the high-level formalism, such that parallel transitions can be modeled, or interleaved ones.

**AbstractMachine** Abstract machines are more elaborate than flows or transitions. They are assumed to have an *internal* state, which has to be updated at regular intervals. As a result, the *external* state denotes whether flows or transitions are enabled or disabled.

The external state consists of a set of transitions which are enabled, as well as a flag that denotes if the passing of time is admissible wrt. the internal state. This distinction is made, because flows can only be executed for the complete model, since we assume a *global time*, and flows evolve wrt. time. Nevertheless, the abstract machine may activate or deactivate the set of currently enabled flows, which may change because of its internal state.

The abstract machines' internal state and therefore its behavior depends on the high-level formalism. For example, a concept of partitioning the high-level model state into discrete locations could be encoded within.

The external state is always determined from the internal state, i.e. an abstract machine may indicate whether in the current internal state only transitions, only flows, or one of both may be performed.

We expect that in every conceivable high-level formalism the execution of flows or transitions is mutually exclusive. Otherwise racing conditions might prevent the discrete change of observables due to simultaneous changes by flows. Therefore, the abstract machine's external state can be seen as a constraint for the complete system's execution. An example for high-level formalisms based on the maximal progress concept, or for high-level formalisms allowing the definition of urgent transitions, is the disabling of flows: Whenever an urgent transition is enabled, it is required that the transition has to be executed before time passes. Therefore, the abstract machine prevents the system from taking continuous steps. This implies that (model) time cannot evolve.

The responsibilities of abstract machines are:

*init()* The abstract machine initializes its internal state.

*update()* The purpose of this operation is to update the abstract machine's internal state, along with its external state. It is activated for all abstract machines whenever global state encoded in channels has been potentially modified, i.e. after transitions or flows have been performed.

This operation thus determines the current set of enabled transitions and the current value of the flow flag, depending on the internal pre-state as well as the global state.

*notifyTrans()* The abstract machine updates its internal state, corresponding to the execution of a specific associated transition. For example, if the internal state encodes locations, the execution of a particular transition probably implies a change between locations inside the respective abstract machine, therefore this operation is activated.

**InterfaceModule** Interface modules are software components which abstract from hardware<sup>2</sup> interfaces.

Interface modules are treated in a similar fashion as flows; from the model perspective, they are interpreted as discretized continuous evolutions of values. They are scheduled with fixed frequency and perform an abstraction from raw data received on hardware interfaces to channel data and vice versa. Since hardware interfaces are sources and/or sinks of data in a natural way, sending and receiving of data is distinguished here:

*poll()* This reads data from the associated interface and places it into a corresponding channel. Polling of data is associated with publication times, such that it is ensured that the data will have been distributed to all recipients before it becomes visible.

*transmit()* This sends data to the associated (hardware) interface. In contrast to *poll()*, each interface module retrieves the current data when scheduled and transmits this data immediately to its hardware interface.

**Selector** The selector is the single component that coordinates the abstract machines' external states. It selects transitions and/or flows, when it is activated:

*getSelection()* The selector chooses a set of transitions and sets a single flow-enabling flag, on the basis of the external states of the abstract machines. The set of transitions is the result of a selection procedure among all possible transitions offered by the abstract machines in their current state, such that all selected transitions shall be executed subsequently. The selector must therefore ensure, that no conflicting transitions exist within this selection. For high-level formalisms with a notion of nondeterminism, each (sequential) abstract machines may offer more than one transition for selection, and it is the selector's responsibility to make the choice.

Further, the set of transitions can be empty, which means that no discrete

---

<sup>2</sup>Although the main purpose of interface modules is the abstraction from *hardware* interfaces, technically also software components can be represented.

step can be taken. Since the possibility of a subsequent flow step is controlled by the flow flag, the selector’s choice also defines how to proceed with the execution – by a discrete or flow step.

*init()* The selector may have an internal state, which is initialized with this operation. Additionally, initialization constraints can be checked on the state space; the operation is expected to give a boolean result that denotes if the state space is initially well-formed or not.

The selector is different from the other objects in that it shall not be instantiated for the particular high-level model, but for the high-level *formalism*. Therefore, every model-specific control behavior must be encoded into the abstract machines of the HL<sup>3</sup> representation.

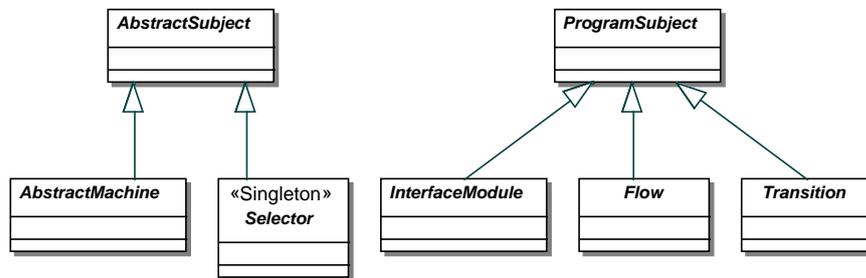


Figure 4.4: Executable objects are grouped into *abstract subjects* and *program subjects*.

The active objects of an HL<sup>3</sup> model are also called *subjects*. We distinguish two different kinds of subjects – (1) Abstract machines and the selector can be arbitrarily complex, therefore abstract definitions of their functionality are assumed. They are *abstract subjects*. (2) Flows, transitions, and interface modules are supposed to be less elaborate. Particularly, they have no internal state, such that they can be given by while-programs. These are called *program subjects*.

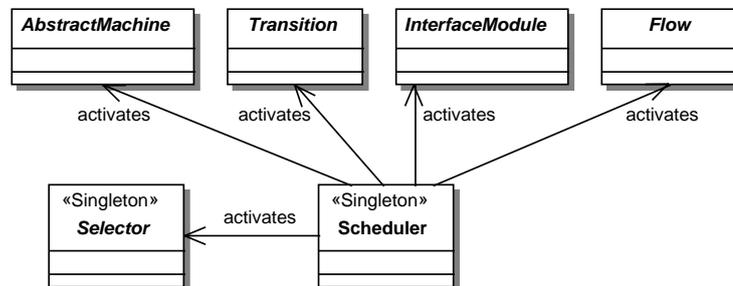


Figure 4.5: The HL<sup>3</sup> Scheduler schedules the active objects of the design pattern.

## 4.2 CONST – Constant State Components

An element  $c \in CONST$  of the constant part of the state space  $S$  defines one particular HL<sup>3</sup> model. It consists of (1) the active and passive entities described in section 4.1, (2) structural aspects of the high-level model, as well as (3) physical constraints specifications.

Formally, the constant state components are given by the set

$$\begin{aligned}
CONST = & \{scheduler\} \times SELECTOR \times \\
& \mathcal{P}(AM) \times \mathcal{P}(IFM) \times \mathcal{P}(FLOW) \times \mathcal{P}(TRANS) \times \\
& \mathcal{P}(VAR) \times \mathcal{P}(CHAN) \times \mathcal{P}(PORT) \times \mathcal{P}(LWP) \times \\
& Am_{Trans} \times Am_{Flow} \times Subject_{Var} \times \\
& Chan_{Port} \times InitVal_{Port} \times Subject_{Port} \times SelPort \times \\
& VisibilitySet_{Flow} \times VisibilitySet_{Ifm} \times VisibilitySet_{Trans} \times \\
& Lwp_{Subj_{abs}} \times Program_{Subj_{prog}} \times \\
& SysPeriod \times Period_{Flow} \times Period_{Ifm,poll} \times Period_{Ifm,tmit}
\end{aligned}$$

The detailed description is given below. For convenience, named projections<sup>3</sup> are given. The following mapping names are available for access to the coordinates of  $c \in CONST$ :

$$\begin{aligned}
sched &= \pi_1 CONST, sel = \pi_2 CONST, am = \pi_3 CONST, \\
ifm &= \pi_4 CONST, flow = \pi_5 CONST, trans = \pi_6 CONST, \\
var &= \pi_7 CONST, chan = \pi_8 CONST, port = \pi_9 CONST, \\
lwp &= \pi_{10} CONST, am_{trans} = \pi_{11} CONST, am_{flow} = \pi_{12} CONST, \\
subject_{var} &= \pi_{13} CONST, chan_{port} = \pi_{14} CONST, initval_{port} = \pi_{15} CONST, \\
subject_{port} &= \pi_{16} CONST, selport = \pi_{17} CONST, vis_{flow} = \pi_{18} CONST, \\
vis_{ifm} &= \pi_{19} CONST, vis_{trans} = \pi_{20} CONST, lwp_{subj} = \pi_{21} CONST, \\
prg_{subj} &= \pi_{22} CONST, \delta_{period} = \pi_{23} CONST, period_{flow} = \pi_{24} CONST, \\
period_{ifm,poll} &= \pi_{25} CONST, period_{ifm,tmit} = \pi_{26} CONST
\end{aligned}$$

Similarly, access to the constant state components  $c$  of states  $(c, v) \in S$  is given by  $const_S = \pi_1 S$ .

Only dedicated parts of the constant state space are accessible from the subjects of the HL<sup>3</sup> model, as used in section 4.6. These are

$$CONST_m = Subject_{Var} \times Chan_{Port} \times Subject_{Port}$$

### 4.2.1 Entities

The entities of an HL<sup>3</sup> model are the union of the entities given by the design pattern and runtime environment, as described in section 4.1. Additionally, a set of *light weight processes* is given. This represents the extent of true parallelism that is available for the model's execution.

We define the sets of *possible* entities that can be contained by HL<sup>3</sup> models  $c \in CONST$ . Additionally, we explicitly assume a specific model  $c$  that contains specific sets of entities. Therefore, we can tailor the definitions of dependencies and physical constraints in the subsequent sections to the entities of  $c$ , and can omit all possible entities that are not part of the specific model.

<sup>3</sup>Projections are defined as  $\pi_i X : X_1 \times \dots \times X_n \rightarrow X_i$  with  $(x_1, \dots, x_n) \mapsto x_i$  for  $X = X_1 \times \dots \times X_n$  and  $1 \leq i \leq n$ .

**scheduler** identifies the pre-defined HL<sup>3</sup> scheduler. There is exactly one scheduler for all particular HL<sup>3</sup> models with a fixed behavior, leading to the operational rules given in section 4.4.

**SELECTOR** is the set of selectors that exist for all possible hybrid systems specification formalisms. The selector  $sel(c) \in SELECTOR$  is tailored for the applied high-level formalism, i.e. for the high-level formalism from which the HL<sup>3</sup> model  $c \in CONST$  originates. We abbreviate it as  $selector = sel(c)$ .

**AM** is the set of possible abstract machine identifiers. The set  $am(c) \subseteq AM$  is derived from high-level specification, such that the sequential and parallel control aspects from the high-level specification are implemented by these abstract machines. We use  $Am = am(c)$  as an abbreviation.

**IFM** contains interface modules identifiers. They implement (hardware) interfaces according to the high-level specification. The available interface modules of HL<sup>3</sup> model  $c \in CONST$  are  $Ifm = ifm(c)$  with  $Ifm \subseteq IFM$ .

**FLOW** are flow identifiers. The flows of  $c \in CONST$  which implement the high-level model's time-continuous calculations are  $Flow = flow(c)$  with  $Flow \subseteq FLOW$ .

**TRANS** are transition identifiers. The transitions that represent the discrete steps of the high-level model are  $Trans = trans(c)$  with  $Trans \subseteq TRANS$ .

**VAR** is the set of available variable symbols. Each local variable which is used in the HL<sup>3</sup> model  $c \in CONST$  is contained in  $Var = var(c) \subseteq VAR$ .

**CHAN** is the set of channel identifiers. Each global or shared variable from the high-level model  $c \in CONST$  is represented by a channel in  $Chan = chan(c) \subseteq CHAN$ .

**PORT** provides *port* identifiers for *channel access*: Data is not read or written directly on channels, but through ports. Particularly, subjects that act as data *recipients* are abstracted by the ports they read. Therefore, the addressing of recipients is done in a uniform way. For an HL<sup>3</sup> model  $c \in CONST$ , the available ports are given by  $Port = port(c) \subseteq PORT$ .

**LWP.** The HL<sup>3</sup> has an execution model which explicitly supports *true parallelism*: An HL<sup>3</sup> program is executed on one or more *light weight processes* (LWP). There are specific light weight processes  $Lwp = lwp(c) \subseteq LWP$  for models  $c \in CONST$ . Each LWP runs exclusively on a dedicated CPU, i.e. there is a fixed mapping  $cpu : Lwp \rightarrow Cpu$ . Each LWP executes without any interference of an underlying operating system<sup>4</sup>. Each subject is allocated on one of the available LWPs in order to be executed. If several subjects shall be allocated to the same LWP, only one of them may be active at a time. Active subjects operating on different LWPs are running simultaneously, having simultaneous

<sup>4</sup>An implementation based on the Linux operating system is provided in [Efk05]

access to all resources (memory, interfaces etc.). Subjects running on the same LWP only access resources one after another, since only one of them is active at any point in time.

For the subjects, i.e. the active objects of the HL<sup>3</sup> model, the distinction into *abstract subjects* and *program subjects* is not encoded explicitly, because it is given implicitly:

$$\begin{aligned} Subject &= Subj_{abs} \cup Subj_{prog} \\ Subj_{abs} &= Am \cup \{selector\} \\ Subj_{prog} &= Flow \cup Trans \cup Ifm \end{aligned}$$

### 4.2.2 Dependencies

For the entities of an HL<sup>3</sup> model, static dependencies exist:

**Am<sub>Trans</sub>.** Each transition of the model is exclusively controlled by one abstract machine:

$$Am_{Trans} = Trans \rightarrow Am$$

**Am<sub>Flow</sub>.** The same holds for flows – each one is controlled by a dedicated abstract machine:

$$Am_{Flow} = Flow \rightarrow Am$$

**Subject<sub>Var</sub>.** A local variable is only accessible for one subject:

$$Subject_{Var} = Var \rightarrow Flow \cup Trans \cup Ifm \cup Am$$

**Chan<sub>Port</sub>.** A port provides access to exactly one channel. There is no distinction between read and write access, i.e. always read/write access is provided.

$$Chan_{Port} = Port \rightarrow Chan$$

**InitVal<sub>Port</sub>.** For each port, there is an initial data value which is read before any further value is published for this port through the corresponding channel.

$$InitVal_{Port} = Port \rightarrow Data$$

**Subject<sub>Port</sub>.** Each particular port of a channel is assigned to a set of subjects. This defines the subjects which are allowed to access the port.

$$Subject_{Port} = Port \rightarrow \mathcal{P}(Am \cup Flow \cup Trans \cup Ifm)$$

**SelPort.** The selector may access ports for special purposes. For example, the resetting of signals from a high-level formalism could be the selector's responsibility.

$$SelPort = \mathcal{P}(Port)$$

**VisibilitySet<sub>Flow</sub>.** Each flow has a specific visibility set, which is used to define a maximal set  $R \subseteq Port$  of recipients for the calculation results of the flow. When the flow writes data to a channel  $cn$ , the actual recipients are the ports  $\{p \in Port \mid chan_{port}(c)(p) = cn\} \cap R$ .

The time component  $t$  of the visibility set acts as a constant publication delay per recipient: Basically, before `integrate()` is executed, the scheduler will choose publication times such that the data will be visible for all recipients as soon as time increases. Then, the scheduler adds the specified delays to these publication times. The resulting visibility set is passed as input parameter for `integrate()`.

$$VisibilitySet_{Flow} = Flow \rightarrow VisibilitySet$$

**VisibilitySet<sub>Ifm</sub>.** The visibility sets which are given for interface modules are similar to the flows' visibility sets, i.e. a recipient set  $\{p \in Port \mid chan_{port}(c)(p) = cn\} \cap R$  for the specified recipient set  $R$  results, and the given times are regarded as delays.

$$VisibilitySet_{Ifm} = Ifm \rightarrow VisibilitySet$$

**VisibilitySet<sub>Trans</sub>.** The visibility sets associated with transitions are also similar to the flows' and interface modules' visibility sets, but with a single difference: The basic publication times originate from the selector, rather than from the scheduler, and therefore depend on the selection policy of the applied high-level formalism.

$$VisibilitySet_{Trans} = Trans \rightarrow VisibilitySet$$

**LwpSubj<sub>abs</sub>.** Abstract subjects  $s \in Subj_{abs}$  are statically assigned to LWPs:

$$Lwp_{Subj_{abs}} = Subj_{abs} \rightarrow Lwp$$

This is due to the fact that abstract subjects can have internal state. The HL<sup>3</sup> framework does not constrain the implementation of abstract subjects, therefore it is not guaranteed that the internal state is accessible from arbitrary LWPs. In contrast, program subjects do not have internal state and can be scheduled dynamically to any available LWP.

**ProgramSubj<sub>prog</sub>.** Within the HL<sup>3</sup> framework, program subjects  $s \in Subj_{prog}$  are given in more detail than abstract subjects. Their behavior is explicitly specified by *while-programs*, which are defined wrt. the subject's operations. The available operations are listed in section 4.3.

$$Program_{Subj_{prog}} = (Flow \times Op_{Flow}) \cup (Trans \times Op_{Trans}) \cup (Ifm \times Op_{Ifm}) \\ \rightarrow Program$$

The program is given by a program string. The semantics of program strings is discussed in section 4.6.

### 4.2.3 Physical Constraints

For the representation of a hybrid high-level model in HL<sup>3</sup>, a *discretization* takes place. This is a necessity, because HL<sup>3</sup> models shall be executable on real hardware, which inherently work in a discrete fashion. Therefore, the periods which define the time durations for subsequent calculation steps of continuous evolutions from the model are significant:

**SysPeriod.** The internal scheduling period  $\delta_{period}(c)$  is the smallest time duration which the execution of the HL<sup>3</sup> model can observe. Therefore, all continuous calculations, i.e. flow and interface module activations, have to occur at time intervals which are multiples of  $\delta_{period}(c)$ .

It is a *physical* time duration, i.e.:

$$SysPeriod = PhysicalTime$$

Physical time is defined in section 4.3.

**Period<sub>Flow</sub>.** For a flow  $f$ , value  $period_{flow}(f)$  denotes the multiple of  $\delta_{period}(c)$ , such that  $p_f = \delta_{period}(c) \cdot period_{flow}(f)$  is the flow's scheduling period:

$$Period_{Flow} = Flow \rightarrow \mathbb{N}$$

**Period<sub>Ifm,poll</sub>.** Analogously, function  $period_{ifm,poll}$  defines the scheduling period for the polling of data from interface modules:

$$Period_{Ifm,poll} = Ifm \rightarrow \mathbb{N}$$

**Period<sub>Ifm,tmit</sub>.** Function  $period_{ifm,tmit}$  defines the scheduling period for the transmitting of data to interface modules:

$$Period_{Ifm,tmit} = Ifm \rightarrow \mathbb{N}$$

## 4.3 VAR – Variable State Components

The variable portion *VAR* of the state space  $S$  is structured into the following sub-components:

$$\begin{aligned} VAR &= PhysicalTime \times ModelTime \times FailStatus \times \Sigma_{LWP} \times \\ &Sched \times \Sigma_{Subjprog} \times \Sigma_{Subjabs} \times \Sigma_{Flow} \times \Sigma_{Am} \times \\ &\Sigma_{Var} \times \Sigma_{Chan} \end{aligned}$$

As for *CONST*, named projections are given for *VAR*:

$$\begin{aligned} physTime &= \pi_1 VAR, modelTime = \pi_2 VAR, fail = \pi_3 VAR, \\ \kappa_{LWP} &= \pi_4 VAR, sched = \pi_5 VAR, \kappa_{Subjprog} = \pi_6 VAR, \\ \kappa_{Subjabs} &= \pi_7 VAR, \kappa_{Flow} = \pi_8 VAR, \kappa_{Am} = \pi_9 VAR, \\ \sigma_{Var} &= \pi_{10} VAR, \kappa_{Chan} = \pi_{11} VAR \end{aligned}$$

For states  $(c, v) \in S$ , the projection  $var_S = \pi_2 S$  provides the variable components  $v$ .

For use in section 4.6, the portions of the variable state space that can be read or written, respectively, by the subjects of the HL<sup>3</sup> model are given:

$$\begin{aligned} VAR_{mread} &= ModelTime \times \Sigma_{Var} \times \Sigma_{Chan} \\ VAR_{mwrite} &= \Sigma_{Var} \times \Sigma_{Chan} \end{aligned}$$

In the following, the components of *VAR* are defined and explained. Additionally, constraints on the initial state  $s_0 = (c, v_0)$  of the state transition system  $sts = (S, s_0, T)$  are given, per component.

**PhysicalTime – Global Physical Time.** Physical time *PhysicalTime* is modeled by the non-negative real numbers:

$$PhysicalTime = \mathbb{R}_0^+$$

It is *observed* within an execution of an HL<sup>3</sup> model, exclusively by the HL<sup>3</sup> scheduler. The HL<sup>3</sup> subjects cannot evaluate it.

*Init State.* Physical time is observed relatively to system start:  $physTime(v_0) = 0$

**ModelTime – Logical HL<sup>3</sup> Time.** The logical HL<sup>3</sup> time is modeled by *time ticks* which are pairs of non-negative integral numbers:

$$ModelTime = \mathbb{N}_0 \times \mathbb{N}_0$$

For convenience, named projections are defined:  $t_0 = \pi_1 ModelTime$ ,  $t_1 = \pi_2 ModelTime$

Component  $t_0(t)$  of  $t \in ModelTime$  represents a discretized abstraction of the physical time as visible to the HL<sup>3</sup> subjects. It is always ensured that  $t_0(modelTime(v)) \leq physTime(v)$ , but – depending on the high-level formalism to be encoded in HL<sup>3</sup> –  $t_0(modelTime(v))$  may be kept constant for some interval of  $modelTime(v)$ , in order to simulate the execution of transitions in zero time. The second component  $t_1(t)$  of the logical HL<sup>3</sup> time is used to distinguish causally related events which occur during the same time tick  $t_0(t)$ . Component  $t_1(t)$  is reset when  $t_0(t)$  is increased.

Alternatively to  $(t_0, t_1) \in ModelTime$ , we write  $t_0.t_1 \in ModelTime$ , because the natural ordering of values  $t_0.t_1$  is almost similar to the ordering of real numbers:  $a_0.a_1 \leq b_0.b_1$  iff  $a_0 < b_0 \vee a_0 = b_0 \wedge a_1 \leq b_1$ .

The addition of two time ticks is defined as:

$$\begin{aligned} + : ModelTime \times ModelTime &\rightarrow ModelTime \\ (t_0.t_1, 0.u_1) &\mapsto t_0.t_1 + u_1 \\ (0.t_1, u_0.u_1) &\mapsto u_0.t_1 + u_1 \\ (t_0.t_1, u_0.u_1) &\mapsto t_0 + u_0.0 \text{ if } t_0 \neq 0 \wedge u_0 \neq 0 \end{aligned}$$

*Init State.* Model time starts at zero:  $modelTime(v_0) = 0.0$

**FailStatus – Execution Failure.** For all variants of hard real-time applications, it is important to detect violations of timing restrictions. To record that such a violation has happened, we use set

$$FailStatus = \{ok, failed\}$$

An execution starts in state *ok* and performs a transition to *failed* if any violation of timing restrictions occurs. The detailed conditions for failure are modeled in the operational rules in section 4.4.

In addition to timing failure, the model execution may fail within an explicit initialization phase (see description of **Sched** below) to indicate that there is no valid execution for the init state of the HL<sup>3</sup> model.

*Init State.* Initially, every execution is ok:  $fail(v_0) = ok$

**$\Sigma_{LWP}$  – Light Weight Processes.** State component  $\Sigma_{LWP}$  reflects the present activation state of subjects. It is modeled as a mapping from available LWPs to active subjects, such that for each LWP, the currently allocated subject is given. If an LWP  $l \in Lwp$  has no subject, then  $\Sigma_{LWP}(l) = scheduler$  denotes that the scheduler is in control of  $l$ , in order to schedule some inactive subject or to leave  $l$  idle.

$$\Sigma_{LWP} = Lwp \rightarrow Subj_{abs} \cup Subj_{prog} \cup \{scheduler\}$$

*Init State.* At system start, no subject is running:  $\text{ran } \kappa_{LWP}(v_0) = \{scheduler\}$

**Sched – The Schedule.** When an active subject releases its LWP  $p$ , the scheduler will obtain control of  $p$ . During its activation time, it will examine the *Sched* state component modeling the current state of the HL<sup>3</sup> program schedule.

We assume *symmetric multi-processing* for any implementation of the scheduler, i.e. the logical scheduler is distributed redundantly to all available LWPs. All local schedulers synchronize wrt. the global HL<sup>3</sup> program schedule. The relevant data structure for the operations of the scheduler is

$$\begin{aligned} Sched &= (\mathcal{P}(Am \times Op_{Am}) \cup \mathcal{P}(Trans \times Op_{Trans})) \\ &\quad \cup \mathcal{P}((Flow \times Op_{Flow}) \cup (Ifm \times Op_{Ifm})) \\ &\quad \cup \{(selector, op) \mid op \in Op_{selector}\} \\ &\quad \times sched\_phase \end{aligned}$$

with

$$\begin{aligned} Op_{Am} &= \{init, update, notifyTrans\} \\ Op_{Trans} &= \{action\} \\ Op_{Flow} &= \{integrate\} \\ Op_{Ifm} &= \{poll, transmit\} \\ Op_{selector} &= \{init, getSelection\} \\ Op &= Op_{Am} \cup Op_{Trans} \cup Op_{Flow} \cup Op_{Ifm} \cup Op_{selector} \end{aligned}$$

and

$$\text{sched\_phase} = \{\text{init\_phase}, \text{update\_phase}, \text{selection\_phase}, \text{flow\_phase}, \\ \text{transition\_phase}, \text{notify\_phase}\}$$

We have a set  $\text{subj\_sched} = \pi_1 \text{Sched}$  which provides pairs  $(s, op)$  of subjects and operations, such each subject  $s$  has to be scheduled in order to execute its operation  $op$ . Further,  $\text{phase} = \pi_2 \text{Sched}$  indicates which processing phase of the HL<sup>3</sup> execution model is active. If  $\text{subj\_sched}(\text{sched}(v)) = \emptyset$  in a state  $(c, v) \in \mathcal{S}$ , then there is nothing more to do for the current processing phase, and a switch to the next phase is to follow.

*Init State.* A system execution starts with initialization of abstract machines and the selector:  $\text{sched}(v_0) = ((Am \cup \{\text{selector}\}) \times \{\text{init}\}, \text{init\_phase})$

**$\Sigma_{\text{Subj}_{\text{prog}}}$  – The Execution State of Program Subjects.** The execution state models for each operation of a *program subject* the remaining statements to be executed. Following the standard concepts for explaining operational semantics of sequential programs [AO97], the execution state of one sequential unit is modeled as the string  $\text{prg} \in \text{Program}$  of programming statements  $s \in \text{Stmt}$  still to be performed:

$$\text{Program} = \text{seq Stmt}$$

Since we are modeling executions on real hardware, all atomic statements will complete in finite time. There is an interval  $[\delta_0, \delta_1]$  for each statement with  $\delta_i \in \mathbb{R}_+$ , such that  $\delta \in [\delta_0, \delta_1]$  whenever the associated subject is scheduled and the statement is not yet completed. Therefore the time value  $\delta \in \mathbb{R}_0^+ \cup \{\infty\}$  is attached to the string of programming statements, denoting the remaining execution time needed to process the actual atomic statement. However, while the object is not running, this time value is set to  $\infty$ , until the object is allocated on an LWP.

Finally, a parameter  $p \in \text{Param}_{\text{prog}}$  may be given for the current execution of the program. The only possible parameter for operations of program subjects is a visibility set, which is applied to the operations *action*, *integrate*, and *poll*, on transitions, flows, and interface modules, respectively. The absence of parameters is denoted by  $\lambda$ .

$$\text{Param}_{\text{prog}} = \text{VisibilitySet} \cup \{\lambda\}$$

Then, the execution state of program subjects is defined as

$$\text{ProgState} = \text{Program} \times (\mathbb{R}_0^+ \cup \{\infty\}) \times \text{Param}_{\text{prog}} \\ \Sigma_{\text{Subj}_{\text{prog}}} = \text{Subj}_{\text{prog}} \times \text{Op} \rightarrow \text{ProgState}$$

For convenience, the coordinates of *ProgState* are given by  $\text{string}_{\text{ProgState}} = \pi_1 \text{ProgState}$ ,  $\delta_{\text{ProgState}} = \pi_2 \text{ProgState}$ , and  $\text{vis}_{\text{ProgState}} = \pi_3 \text{ProgState}$ .

*Init State.* At the beginning of a model run, no program subject is running:  $\text{ran } \kappa_{\text{Subj}_{\text{prog}}}(v_0) = \{(\langle \rangle, \infty, \lambda)\}$

**$\Sigma_{\text{Subj}_{\text{abs}}}$  – The Execution State of Abstract Subjects.** The execution state of an *abstract subject* consists of an internal state component  $s \in \text{IntState}$ , as well as a time value  $\delta \in \mathbb{R}_0^+ \cup \{\infty\}$  denoting the remaining execution time needed to process its current task, similarly to the execution of program subjects. The value  $\infty$  denotes that the object is not allocated on an LWP.

Possible execution parameters  $p \in \text{Param}_{\text{abs}}$  are a single transition each, which is needed for the *notifyTrans* operations on abstract machines:

$$\text{Param}_{\text{abs}} = \text{Trans} \cup \{\lambda\}$$

The execution state of abstract subjects is given by

$$\begin{aligned} \text{AbsState} &= \text{IntState} \times (\mathbb{R}_0^+ \cup \{\infty\}) \times \text{Param}_{\text{abs}} \\ \Sigma_{\text{Subj}_{\text{abs}}} &= \text{Subj}_{\text{abs}} \rightarrow \text{AbsState} \end{aligned}$$

For convenience, the coordinates of *AbsState* are given by  $\text{intState}_{\text{AbsState}} = \pi_1 \text{AbsState}$ ,  $\delta_{\text{AbsState}} = \pi_2 \text{AbsState}$ , and  $\text{trans}_{\text{AbsState}} = \pi_3 \text{AbsState}$ .

*Init State.* At the beginning of a model execution, no abstract subject is running:  $\text{ran } \kappa_{\text{Subj}_{\text{prog}}}(v_0) \subseteq \{(s, \infty, \lambda) \in \text{AbsState}\}$ . The internal states  $s$  are undetermined, but will be initialized during the *init\_phase*.

**$\Sigma_{\text{Flow}}$  – The Flow State.** The flow state specifies for each flow whether it is currently enabled, and, if this is the case, the absolute point in physical time for its next periodic execution.

$$\begin{aligned} \text{FlowState} &= \mathbb{R}_0^+ \cup \{\infty\} \\ \Sigma_{\text{Flow}} &= \text{Flow} \rightarrow \text{FlowState} \end{aligned}$$

For  $(c, v) \in \text{VAR}$  and  $f \in \text{Flow}$ ,  $\kappa_{\text{Flow}}(v)(f) = \infty$  indicates that the flow is currently disabled and shall not be scheduled. A value  $\kappa_{\text{Flow}}(v)(f) \in \mathbb{R}_0^+$  indicates that  $f$  shall be scheduled at some time  $t \in \text{PhysicalTime}$  with  $\kappa_{\text{Flow}}(v)(f) \leq t < \kappa_{\text{Flow}}(v)(f) + \delta_{\text{period}}(c)$ . If the scheduler cannot manage to reserve an LWP for  $f$  for such a time, a transition to  $(c, v')$  with  $\text{fail}(v') = \text{failed}$  is performed. A transition into the failure state is also performed if the flow is scheduled correctly at time  $t$ , but terminates later than  $\kappa_{\text{Flow}}(v)(f) + \delta_{\text{period}}(c)$ . That means, that the flow must be also completed within the specified system period.

*Init State.* In the beginning, all flows are disabled:  $\text{ran } \kappa_{\text{Flow}}(v_0) = \{\infty\}$

**$\Sigma_{\text{Am}}$  – The Abstract Machine State.** The abstract machine status indicates its enabledness for continuous or discrete steps:

$$\begin{aligned} \text{AmState} &= \mathbb{B} \times \mathcal{P}(\text{Trans}) \\ \Sigma_{\text{Am}} &= \text{Am} \rightarrow \text{AmState} \end{aligned}$$

with named projections  $\text{flow}_{\text{AmState}} = \pi_1 \text{AmState}$  and  $\text{trans}_{\text{AmState}} = \pi_2 \text{AmState}$ .

*Continuous Step.* The abstract machine  $m \in Am$  indicates, whether it admits a continuous step of the *complete system*. This decision is made internally and depends on the applied high-level formalism.

Since time passes for all abstract machines  $Am$  of the system, a continuous step can only be taken by all abstract machines together. Therefore, each abstract machine indicates its enabledness for this by a boolean flag, such that the conjunction of all flags defines whether a continuous step is possible for the complete system.

Whether the system actually switches to a flow phase is the choice of the scheduler, based on the selector's selection. Which set of flows is executed in such a case is defined by the flow entities themselves.

*Discrete Step.* The abstract machines' status also contains, whether a discrete step is possible. Unlike continuous steps, discrete steps may be executed for a subset  $Am_{dstep} \subseteq Am$  of all abstract machines. A discrete step consists of the execution of a set of transitions.

Transitions  $t$  are either enabled or disabled, depending on the current state  $(c, v) \in S$ . This is chosen internally within the abstract machine  $m = am_{trans}(c)(t)$ , i.e. the abstract machine that is responsible for the particular transition.

For a composition of a discrete step, each abstract machine indicates which of its transitions are available, i.e. it provides the set of its enabled transitions.

The selector determines which enabled transitions are to be fired within a discrete step. Since abstract machines are designed as sequential components, every admissible selector will select at most one transition per abstract machine. The scheduler will then allocate the actions associated with all selected transitions on an LWP and notify the corresponding abstract machines about the transitions which have been fired.

The abstract interface postulated for an abstract machine  $m$  requires that a call to its `update()` method will calculate the set of enabled transitions (as well as the boolean flag for continuous step-enabledness) again. Observe that abstract machines may label more than one transition as enabled if the underlying high-level formalism admits nondeterministic behavior. Moreover, if the formalism contains the concept of urgent transitions, this can be reflected by a non-empty set of enabled transitions in combination with the *disabling* of continuous steps. This causes the selector to select a discrete step, rather than a flow phase.

*Init State.* Abstract machines do not select anything, initially:  $\text{ran } \kappa_{Am}(v_0) = \{(false, \emptyset)\}$

**$\Sigma_{Var}$  – Local Variable Valuations.** Valuations of local variables are specified in the conventional way which is also applied when defining operational semantics to sequential programs: A valuation  $\sigma : Var \rightarrow Val$  maps each local variable symbol  $x \in Var$  to its current value  $\sigma(x) \in type_{Var}(x)$ , i.e. the variables are typed, and  $Val$  is the union of all these types:

$$\begin{aligned} \Sigma_{Var} &= Var \rightarrow Val \\ type_{Var} &: Var \rightarrow \mathcal{P}(Val) \end{aligned}$$

*Init State.* There is no initial valuation for local variables. Therefore, every program that uses local variables must initialize them explicitly.

**$\Sigma_{\text{Chan}}$  – HL<sup>3</sup> Channels.** The HL<sup>3</sup> framework supports a notion of visibility and publication of variable values, based on the concept of HL<sup>3</sup> channels. This is modeled as the current state  $\Sigma_{\text{Chan}}$  of channels.

$$\begin{aligned} \text{ChanState} &= \text{ModelTime} \times \text{Port} \leftrightarrow \text{Data} \\ \Sigma_{\text{Chan}} &= \text{Chan} \rightarrow \text{ChanState} \end{aligned}$$

When inserting a data item into a channel, a *visibility set* is specified for this item. A visibility set is a collection of visibilities, that are time ticks with an associated recipient:

$$\begin{aligned} \text{Visibility} &= \text{ModelTime} \times \text{Port} \\ \text{VisibilitySet} &= \mathcal{P}(\text{Visibility}) \end{aligned}$$

For convenience, we define an operation  $\bowtie$  that builds the intersection of two visibility sets wrt. the recipients, and adds the corresponding time ticks:

$$\begin{aligned} \bowtie : \text{VisibilitySet} \times \text{VisibilitySet} &\rightarrow \text{VisibilitySet} \\ (v_1, v_2) &\mapsto \{(t, p) \in \text{ModelTime} \times \text{Port} \\ &\mid \exists (t_1, p) \in v_1, (t_2, p) \in v_2 \bullet t = t_1 + t_2\} \end{aligned}$$

When data is retrieved from a channel through a port  $p \in \text{Port}$ , access to either the most recent data item visible for  $p$ , or to the second most recent item is provided. Thus, visibility sets  $v \in \text{VisibilitySet}$  are used to specify *when* data values should become visible to specific recipients.

The types of application-specific data are transparent to the HL<sup>3</sup> framework, since the data is never interpreted or changed by operational rules of the framework. Therefore application-specific data are modeled just as sequences of bytes:

$$\text{Data} = \text{seq Bytes}$$

We assume that there is a unique byte representation for every application-specific data value:

$$\begin{aligned} \text{data}_{\text{Val}} : \text{Val} &\rightsquigarrow \text{Data} \\ \text{val}_{\text{Data}} &= \text{data}_{\text{Val}}^{-1} \end{aligned}$$

*New Entry Insertion.* A new data value is inserted into a channel through a port, such that for each receiving port from the visibility set, an entry is created with the according visibility time:

$$\begin{aligned} \text{ChanEntry} &= \text{Data} \times \text{VisibilitySet} \\ \text{insert}_{\text{Port}} : \text{Port} \times \text{ChanEntry} \times \text{CONST}_m \times \text{VAR}_{m\text{read}} \\ &\rightarrow (\text{Chan} \times \text{ChanState}) \end{aligned}$$

$$\begin{aligned}
& (p_{write}, (data, vis), c, v) \mapsto \\
& \quad (chan_{port}(c)(p_{write}), \\
& \quad \kappa_{Chan}(v)(chan_{port}(c)(p_{write})) \oplus \\
& \quad \{ (tick, p_{read}) \mapsto data \mid (tick, p_{read}) \in vis \})
\end{aligned}$$

Only one data value is stored for each combination  $(t, p)$  of visibility time and recipient port. Therefore, racing conditions take effect on writing into the channel, if data is written concurrently for  $(t, p)$ . This is the desired behavior for the HL<sup>3</sup> semantics. It is the responsibility of the applied high-level formalism to avoid this, if it is required.

*Current Entry Access.* Access to the currently visible entry in a channel is defined by:<sup>5</sup>

$$\begin{aligned}
& entry_{Port,cur} : Port \times CONST_m \times VAR_{mread} \mapsto ModelTime \times Data \\
& (p, c, v) \mapsto (t_{cur}, \kappa_{Chan}(v)(chan_{port}(c)(p))(t_{cur}, p)) \\
& \text{with } t_{cur} = \\
& \quad \mu(t : ModelTime \mid t \leq modelTime(v) \\
& \quad \wedge (t, p) \in \text{dom } \kappa_{Chan}(v)(chan_{port}(c)(p)) \\
& \quad \wedge (\forall u : ModelTime \bullet (u, p) \in \text{dom } \kappa_{Chan}(v)(chan_{port}(c)(p)) \\
& \quad \Rightarrow u \leq t \vee modelTime(v) < u))
\end{aligned}$$

The above mapping retrieves a pair  $(t_{cur}, data) \in ModelTime \times Data$  from the channel  $chn \in Chan$  that is associated with the given port  $p \in Port$ . The system's state  $(c, v) \in S$  determines the current valuation  $\kappa_{Chan}(v)(chn)$  of the channel, as well as the (constant) dependency  $chan_{port}(c)(p)$  between port and channel.  $(t_{cur}, data)$  is chosen such that  $t_{cur}$  is the dedicated time stamp that (1) is not in the (model) future, that (2) actually appears in the channel for the port, and that (3) is the newest of such time stamps.

Access to the currently visible data and visibility time is then given by

$$\begin{aligned}
& data_{Port,cur} : Port \times CONST_m \times VAR_{mread} \mapsto Data \\
& (p, c, v) \mapsto \pi_2(ModelTime \times Data)(entry_{Port,cur}(p, c, v)) \\
& tick_{Port,cur} : Port \times CONST_m \times VAR_{mread} \mapsto ModelTime \\
& (p, c, v) \mapsto \pi_1(ModelTime \times Data)(entry_{Port,cur}(p, c, v))
\end{aligned}$$

*Previous Entry Access.* In addition to the access to the currently visible entry, channels permit reading of the *previous* entry:

$$\begin{aligned}
& entry_{Port,prev} : Port \times CONST_m \times VAR_{mread} \mapsto ModelTime \times Data \\
& (p, c, v) \mapsto (t_{prev}, \kappa_{Chan}(v)(chan_{port}(c)(p))(t_{prev}, p)) \\
& \text{with } T_{prev} = \{t \in ModelTime \mid \\
& \quad \exists t_{cur} \in ModelTime \bullet t < t_{cur} \leq modelTime(v) \\
& \quad \wedge \{(t, p), (t_{cur}, p)\} \subseteq \text{dom } \kappa_{Chan}(v)(chan_{port}(c)(p))\}
\end{aligned}$$

<sup>5</sup>We use the notation  $q = \mu(x : X \mid p)$  for a set  $X$ , a bound variable  $x$  and a predicate  $p$  over  $x$  as a shorthand for  $|\{x \in X \mid p\}| = 1 \Rightarrow q \in \{x \in X \mid p\}$ . Further,  $q = \mu(X)$  is an abbreviation for  $q = \mu(x : X \mid true)$ .

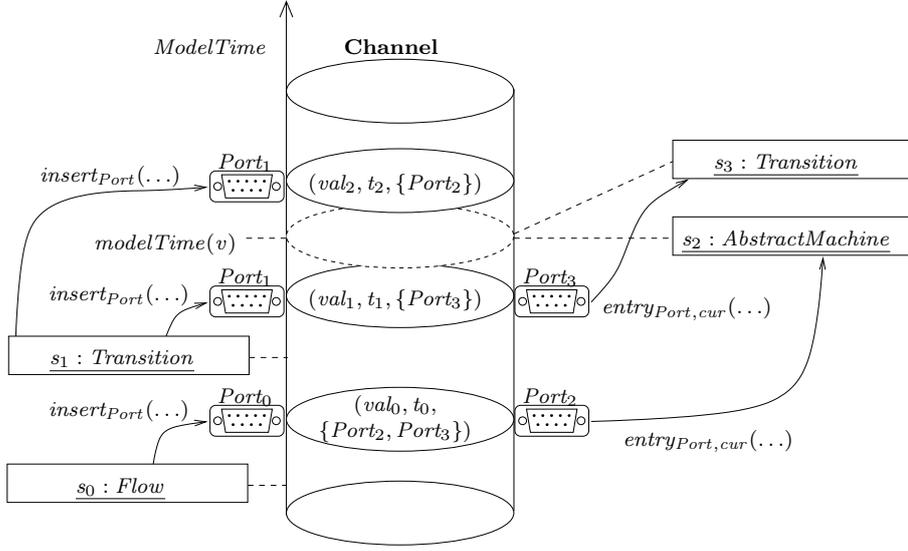


Figure 4.6: Illustration of an HL<sup>3</sup> Channel – subjects  $s_0, s_1$  have written values with different visibility sets to the channel, therefore in state  $(c, v) \in S$ , subjects  $s_2, s_3$  receive different values.

$$\begin{aligned}
 & \wedge (\forall u : ModelTime \bullet (u, p) \in \text{dom } \kappa_{Chan}(v)(chan_{port}(c)(p)) \\
 & \quad \Rightarrow u \leq t \vee modelTime(v) < u) \\
 & \wedge T_{prev} \neq \emptyset \Rightarrow t_{prev} = \mu(T_{prev}) \\
 & \wedge T_{prev} = \emptyset \Rightarrow t_{prev} = tick_{Port,cur}(p, c, v)
 \end{aligned}$$

This mapping provides an entry  $(t_{prev}, data)$  such that  $t_{prev}$  is the dedicated time stamp from the model presence or past that is the *second newest* of all time stamp occurrences for the given port. Alternatively, iff there is only  $t_{cur}$  available (as defined above), this is chosen.

Similarly to  $entry_{Port,cur}$ , the projections to data and time components are given:

$$\begin{aligned}
 data_{Port,prev} & : Port \times CONST_m \times VAR_{mread} \mapsto Data \\
 (p, c, v) & \mapsto \pi_2 (ModelTime \times Data)(entry_{Port,prev}(p, c, v)) \\
 tick_{Port,prev} & : Port \times CONST_m \times VAR_{mread} \mapsto ModelTime \\
 (p, c, v) & \mapsto \pi_1 (ModelTime \times Data)(entry_{Port,prev}(p, c, v))
 \end{aligned}$$

*Init State.* At system start, every channel contains exactly the initial values for its associated ports, with time stamp zero:  $\kappa_{Chan}(v_0) = \{c \mapsto \{(0.0, p) \mapsto initval_{port}(c)(p) \mid p \mapsto c \in chan_{port}\} \mid c \in Chan\}$

## 4.4 Scheduling Rules

In this section, a high-level view on the system execution is given. First, it is described informally, how the assignment of subjects to LWPs evolves wrt. time,

in order to perform their respective tasks – i.e. the *scheduling* is given. This is structured into *scheduling phases* (also called *execution phases*), which are roughly divided into discrete and flow phase, as well as additional housekeeping.

Second, operational rules define the scheduling formally. This includes the switching between complete phases as well as the scheduling of single subjects within phases. The rules that determine the subjects' internal operation are provided in the subsequent sections.

**Scheduling Intuition.** There are two different aspects of scheduling for an HL<sup>3</sup> system: (1) Conceptually, the system resides within an execution loop, that sequentially steps through scheduling phases. Within each iteration, exactly one of *flow phase* or *discrete phase* is chosen. (2) With respect to *physical time*, there is a fixed period  $\delta_{period}(c)$  (for all states  $(c, v) \in S$ ) that defines the points in time for which the (discretized) flow calculations have to be started. This restricts and *requires* when flow phases must occur. As long as the system's choice for flow phases within the execution loop is synchronized with the period  $\delta_{period}(c)$ , i.e. the choice for a flow phase is always in time, the system's timing is ok – *fail(v) = ok*. Otherwise, a timing failure is detected, i.e. *fail(v) = failed*.

**Execution Loop.** An iteration of the system's execution loop starts (1) in phase *update\_phase*, followed by (2) *selection\_phase*. Based on the resulting selection, (3) a choice between *flow\_phase* and *transition\_phase* is made. (4) A *notify\_phase* is appended, iff previously a *transition\_phase* was chosen, otherwise skipped:

*update\_phase* During this phase, the decision for the next flow or discrete step is made. Since the control logic of the system is coded into the set of available abstract machines  $am(c)$ , all of them update their *internal state* and provide their resulting choice for discrete and flow steps (alias *abstract machine state*)  $s \in AmState$ .

*selection\_phase* According to the high-level formalism, a resulting set of transitions and a flag denoting flow-enabledness is determined from the abstract machine's states. From this, either *flow\_phase* or *transition\_phase* is chosen for the next phase.

*flow\_phase* At the beginning of this phase, the synchronization with the physical time is done – the system *waits* for the expiration of the current system period. Afterwards, all flows  $f \in flow(c)$  which are (1) enabled and (2) for which the current time fits to their respective period are recalculated. Interface modules  $md \in ifm(c)$  are treated similarly, they are triggered to put/get data to/from their respective interface.

*transition\_phase* This phase executes all discrete actions that were given by the transitions selected in phase *selection\_phase*.

*notify\_phase* Following phase *transition\_phase*, during this phase each abstract machine for which a transition was selected, is notified. This is necessary, because the firing of a transition  $t \in trans(c)$  can have an impact on the internal state of the associated abstract machine  $m \in am_{trans}(v)(t)$ . Each notified abstract machine then adjusts its internal state correspondingly.

Before the first iteration of the execution loop, i.e. at the start of the model execution, an *init\_phase* is executed exactly once:

*init\_phase* Within this phase, the subjects which contain internal state are initialized, such that they have a defined internal state afterwards. The affected subjects are the abstract machines and the selector.

**Illustrated Scheduling Example.** An example run of an  $HL^3$  system for  $|lwp(c)| = 3$ , i.e. for a system which has three CPUs for parallel execution of subjects, is given in figures 4.7, 4.8, and 4.15. Each figure shows one of three subsequent periods, showing the evolution of physical time from top to bottom. The vertical extent of the boxes denoting the execution of the subjects corresponds to the execution duration of the respective subject. The vertical space between boxes is just for clearness of the presentation, it shall not denote time durations. The horizontal position assigns each execution to one of the available LWPs, the horizontal extent has no meaning. Different phases are separated by dashed lines.

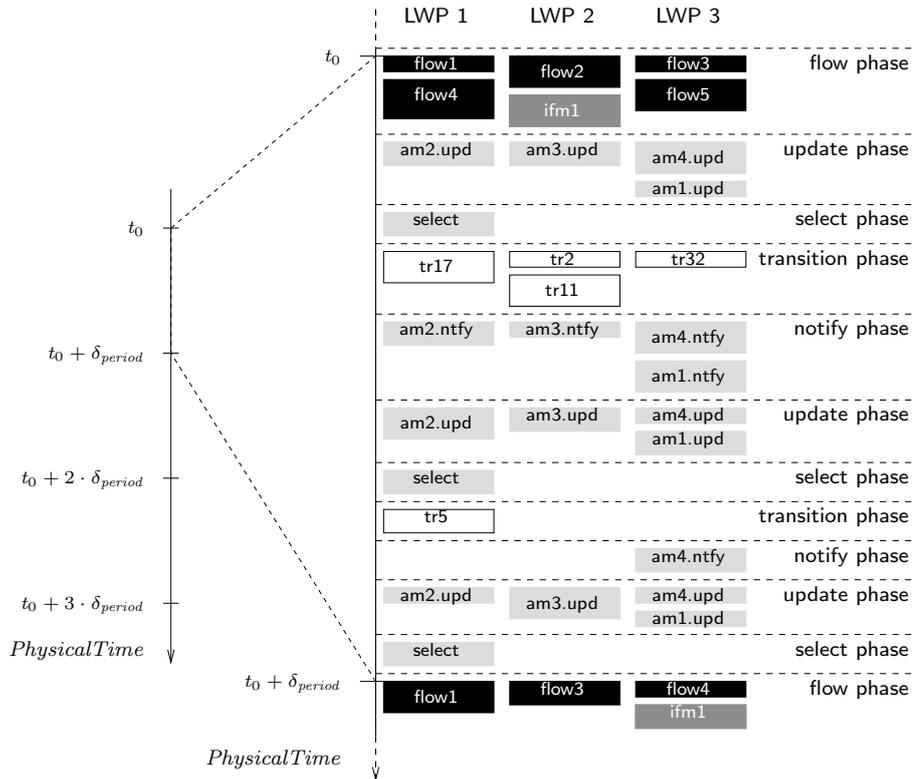


Figure 4.7: Scheduling example: between two consecutive flow phases, two transition phases (along with notification phases) are scheduled.

In figure 4.7, a successful execution of two consecutive flow phases as well as of two transition phases in between is shown: (1) The first flow phase shows the scheduling of flows and interface modules. Each flow or interface module is immediately executed once; whenever an idle LWP is found, one of the remaining

subjects is assigned and started. (2) Next, a new iteration of the execution loop starts with the update phase. All abstract machines  $m$  are scheduled for update, each one on the statically assigned LWP  $p = lwp_{subj}(c)(m)$ . (3) Then, the selector selects the succeeding phase and determines the subjects to be scheduled within. Since this is a centralized activity, only one LWP is active, the others are idle. (4) The selector has decided to continue with a transition phase, therefore all actions of the transitions of the selection are executed. Like for flows, each transition is assigned to the first idle LWP found. (5) Afterwards, the notification of abstract machines is done. Again, the assignment of abstract machines and LWPs is fixed. Here, all abstract machines are affected, because for each one, a transition was taken.

Further, a second execution loop iteration with transition phase is shown. Since only one transition is chosen, there is only one abstract machine to notify. The third iteration chooses a flow phase, therefore the timing wrt. the period is ok. Note that this choice is determined from the abstract machines and the selector, i.e. from the model, and cannot be enforced by the HL<sup>3</sup> environment.

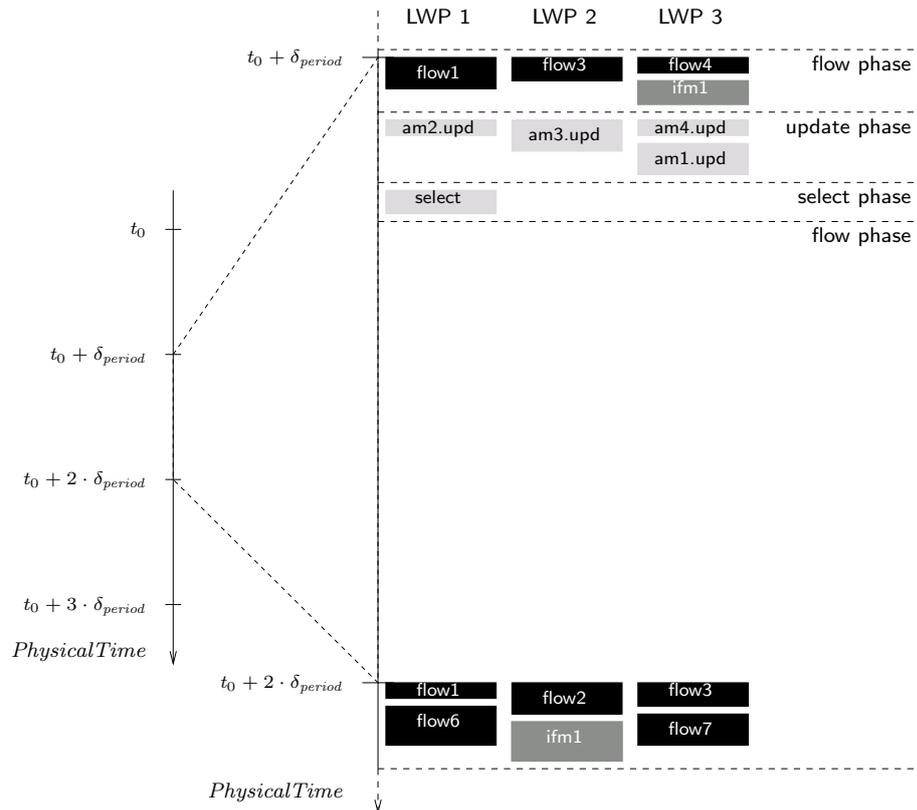


Figure 4.8: Scheduling example: two consecutive flow phases, but no transition phases in between.

The successive period is shown in figure 4.8. Here, the first execution loop iteration chooses a flow phase. Therefore, the complete system is idle for the rest of this period, since flows take place in the beginning of the following period.

Finally, in figure 4.15 a scenario is given that violates the timing: the third iteration of the execution loop chooses a transition phase again, which is yet active when the point in time is reached at which the next flow phase should have been scheduled.

#### 4.4.1 Timing

Since the HL<sup>3</sup> framework is designed for hard real-time systems, timing is a key issue for the execution of an HL<sup>3</sup> model. For arbitrary HL<sup>3</sup> models, it cannot be guaranteed that every run meets its timing requirements, because it depends on the transformation from the high-level formalism on the one hand, and on the particular high-level model on the other. Therefore, it is important that during execution, violations of timing requirements are detected.

The purpose of this section is to *identify* the timing constraints that must hold for states  $s \in S$  for a successful HL<sup>3</sup> model execution. Correspondingly, the state space  $S$  is partitioned into *valid states* and *fail states*.

#### Discretization of Time

For the execution of an HL<sup>3</sup> model  $c \in CONST$ , physical time is *discretized*. In the states of the operational semantics  $(c, v) \in S$ , the physical time itself is contained as  $physTime(v) \in PhysicalTime$ , whereas the model time  $modelTime(v) \in ModelTime$  is the discretized notion of time. The first component  $t_0(t)$  of  $t \in ModelTime$  corresponds to physical time in that the operational semantics will set  $t_0(modelTime(v)) = \lfloor physTime(v) \rfloor$  at dedicated synchronization points in physical time. The second model time component  $t_1(t)$  of  $t \in ModelTime$  is not relevant for the considerations of this section and therefore not discussed here.

Along with the discretized evolution of time, continuous calculations represented by flow and interface module operations  $(Flow \times Op_{Flow}) \cup (Ifm \times Op_{Ifm})$  have to be computed discretely. An ideal discretization would synchronize the model time and would calculate operations in zero time, exactly for multiples of the system period  $\delta_{period}(c)$ . Of course, a real discretized execution needs some time duration for calculations, and fortunately, there is time – between each two consecutive multiples  $n \cdot \delta_{period}(c)$  and  $(n + 1) \cdot \delta_{period}(c)$  of the system period, the time duration  $\delta_{period}(c)$  can be spent for this.

**Physical Time Frame.** Therefore, physical time is partitioned into fixed-sized *frames* of duration  $\delta_{period}(c)$ , beginning with  $t = 0$  for system start. Each frame  $frame(c, n) = (t_{begin}, t_{end})$  defines a physical time interval  $[t_{begin}, t_{end}]$ , and frames are numbered by  $n \in \mathbb{N}_0$ :

$$\begin{aligned} frame &: CONST \times \mathbb{N}_0 \rightarrow PhysicalTime \times PhysicalTime \\ (c, n) &\mapsto (n \cdot \delta_{period}(c), (n + 1) \cdot \delta_{period}(c)) \end{aligned}$$

For convenience, we define  $low = \pi_1(PhysicalTime \times PhysicalTime)$  and  $up = \pi_2(PhysicalTime \times PhysicalTime)$  to access the lower and upper bound of the intervals, respectively.

Then, for each time value  $t$ , the frame number  $n$  for which  $t \in [low(frame(c, n)), up(frame(c, n))]$  holds, is given by

$$fno_{time} : CONST \times PhysicalTime \rightarrow \mathbb{N}_0$$

$$(c, t) \mapsto \left\lfloor \frac{t}{\delta_{period}(c)} \right\rfloor$$

**Flow Execution Interval.** The major uncertainty for the execution of the HL<sup>3</sup> model is the amount of time needed for the calculations within (and between) discretized steps. This cannot be predetermined in general, because abstract machines control dynamically the enabledness of calculations, in a way that depends on the specific high-level model and its specific transformation into HL<sup>3</sup>.

For every frame, a set of flow operations and interface module operations have to be executed. A superset is predefined by the period factors  $period_{flow}(c)$ ,  $period_{ifm,poll}(c)$ , and  $period_{ifm,tmit}(c)$ , but flows can be dynamically disabled by abstract machines. Depending on the number of available LWPs, on the execution times of the calculations, and of their exact activation times, a *flow execution interval* results for each frame number, such that all corresponding operations are executed within:

$$\begin{aligned} fexecint &: CONST \times \mathbb{N}_0 \rightarrow PhysicalTime \times PhysicalTime \\ \forall(c, n) \in CONST \times \mathbb{N}_0 &\bullet low(fexecint(c, n)) \geq low(frame(c, n)) \end{aligned}$$

Note that the flow execution interval also does not end before its frame:

$$\forall(c, n) \in CONST \times \mathbb{N}_0 \bullet up(fexecint(c, n)) \geq low(frame(c, n))$$

The operational semantics will not execute the operations of a flow execution interval *before* the respective frame has begun: The flow execution interval is embedded into the scheduling phase *flow\_phase*, such that their end times coincide. However, the *flow\_phase* generally begins earlier than the flow execution interval, such that the system execution *waits* for its beginning.

**Model Time Synchronization.** Directly after the executions of the operations within the flow execution interval are finished, the operational semantics will synchronize the model time wrt. the physical time. The synchronization points in physical time as they are observed for a particular HL<sup>3</sup> model execution, are given for the corresponding frame number:

$$\begin{aligned} synctime &: CONST \times \mathbb{N}_0 \rightarrow PhysicalTime \\ (c, n) &\mapsto up(fexecint(c, n)) \end{aligned}$$

**Successful Discretization.** The discretization is successful for the execution of an HL<sup>3</sup> model  $c \in CONST$ , iff for every frame number, the model time synchronization takes place within that frame:

$$\begin{aligned} \forall(c, n) \in CONST \times \mathbb{N}_0 &\bullet \\ synctime(c, n) &\in [low(frame(c, n)), up(frame(c, n))] \end{aligned}$$

Therefore, a successful model execution will have exactly one time synchronization per frame. Because we assume (see above) that flow execution intervals do not start too early, it suffices to monitor that a least one synchronization exists per frame:

$$\begin{aligned} \forall(c, n) \in CONST \times \mathbb{N}_0 &\bullet \exists m \in \mathbb{N}_0 \bullet \\ synctime(c, m) &\in [low(frame(c, n)), up(frame(c, n))] \end{aligned}$$

### Monitoring of Successful Discretization

In order to monitor the success or failure of discretization during an HL<sup>3</sup> model execution, we define a constraint on the state space  $S$  of the operational semantics. Since we observe states  $(c_{ob}, v_{ob}) \in S$  during model execution, only those states are accessible that reflect the current physical time  $t$ , i.e.  $physTime(v_{ob}) = t$ . The model time  $modelTime(v_{ob})$  then reflects the most recent time from the past at which a time synchronization has taken place.

Because the time synchronization is monitored during the *complete* execution, it is not necessary to consider all past frames. In fact, only for the last elapsed frame, the time synchronization has to be checked. All preceding frames have been checked in the past already. Therefore,

$$\begin{aligned} \exists m \in \mathbb{N}_0 \bullet & synctime(c_{ob}, m) \in \\ & [low(frame(c_{ob}, fno_{time}(physTime(v_{ob})) - 1)), \\ & up(frame(c_{ob}, fno_{time}(physTime(v_{ob})) - 1))] \end{aligned}$$

must always hold.

For successful executions, the model time  $modelTime(v_{ob})$  always reflects a synchronization time from either the current frame with number  $fno_{time}(physTime(v_{ob}))$ , or the last elapsed frame  $fno_{time}(physTime(v_{ob})) - 1$ . The current frame is not relevant, because by definition, it is not finished yet, and time synchronization may occur within the frame in the future. However, the detection of a time synchronization for the current frame must not denote a failure, therefore we relax the constraint to

$$\begin{aligned} \exists m \in \mathbb{N}_0 \bullet & synctime(c_{ob}, m) \in \\ & [low(frame(c_{ob}, fno_{time}(physTime(v_{ob})) - 1)), \\ & up(frame(c_{ob}, fno_{time}(physTime(v_{ob}))) \end{aligned}$$

Then, when checking this constraint at some time  $physTime(v_{ob}) > low(frame(c, fno_{time}(physTime(v_{ob}))))$ , i.e. *after* the current frame has begun, it can be satisfied even if the last elapsed frame has failed to synchronize time. But directly at the frame's beginning, i.e. at points in time  $physTime(v_{ob}) = low(frame(c, fno_{time}(physTime(v_{ob}))))$ , this failure is detected.

In order to check the constraint, the value of  $modelTime(v_{ob})$  can be used to check it wrt. the given interval, because its first component directly encodes the existence of a corresponding time synchronization (see above):

$$\begin{aligned} t_0(modelTime(v_{ob})) \in \\ & [low(frame(c_{ob}, fno_{time}(physTime(v_{ob})) - 1)), \\ & up(frame(c_{ob}, fno_{time}(physTime(v_{ob}))) \end{aligned}$$

We define a mapping that checks this constraint on states  $(c, v) \in S$  and explicitly states whether  $(c, v)$  is *synchronized* or not. Thereby, the expression is expressed directly, i.e. by resolving the definitions of *frame* and *fno<sub>time</sub>*:

$$\begin{aligned} chktsync : S & \rightarrow \mathbb{B} \\ (c, v) & \mapsto \left\lfloor \frac{physTime(v)}{\delta_{period}(c)} \right\rfloor - 1 \leq \frac{t_0(modelTime(v))}{\delta_{period}(c)} < \left\lfloor \frac{physTime(v)}{\delta_{period}(c)} \right\rfloor + 1 \end{aligned}$$

**Discretization Examples.** Figure 4.9 illustrates a successful synchronization sequence. For all physical times  $t$  in a frame  $fno_{time}(c, t)$ , the model time is either a time of the preceding frame  $fno_{time}(c, t) - 1$ , or of the current frame  $fno_{time}(c, t)$ .

In contrast, a scenario is given in figure 4.10, with synchronization missing in frame  $fno_{time}(c, t) = 3$ . This denotes a model execution that misses a discretization step and therefore fails to execute some continuous calculations in time.

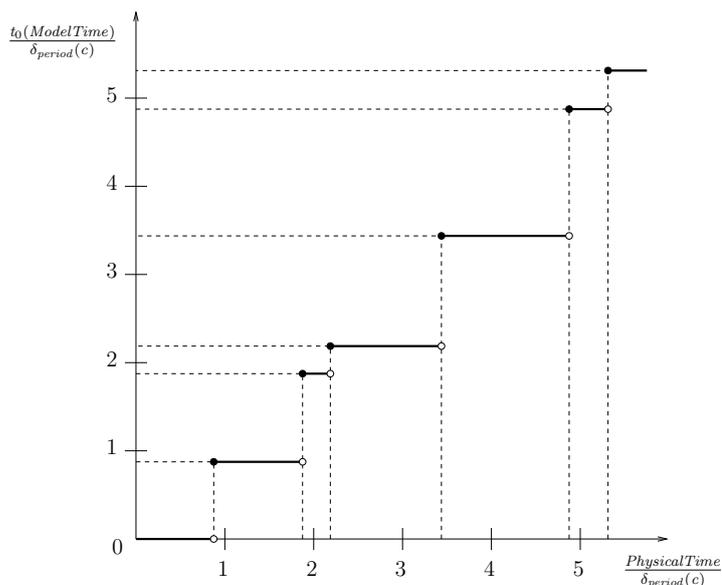


Figure 4.9: Model time must be synchronized within each frame for a successful execution of the HL<sup>3</sup> model. The model time *is* the time of synchronization, therefore it identifies the frame of synchronization.

A successful scheduling scenario is shown in figure 4.11: Calculations of all phases are fast enough. Different shades of grey denote the idle part of the flow phase (light) and the flow execution interval (dark). A failure scenario is given in figure 4.14: The flow phase (and flow execution interval) is started in time, but takes too much time.

We have left out the discussion of non-flow phases in this section, because they affect the discretization of time only in a straight-forward way: The set of execution phases *between* two consecutive flow phases can also take too much physical time. This is illustrated in figure 4.13.

### Time-Triggeredness

As a further restriction on admissible runs of an HL<sup>3</sup> model, we require that every flow execution interval must be started with the beginning of its frame:

$$\forall(c, n) \in CONST \times \mathbb{N}_0 \bullet low(fexecint(c, n)) = low(frame(c, n))$$

See for example figures 4.12 and 4.15: Wrt. to the given constraints, these scenarios would be successful runs of an HL<sup>3</sup> model. Nevertheless, we disallow

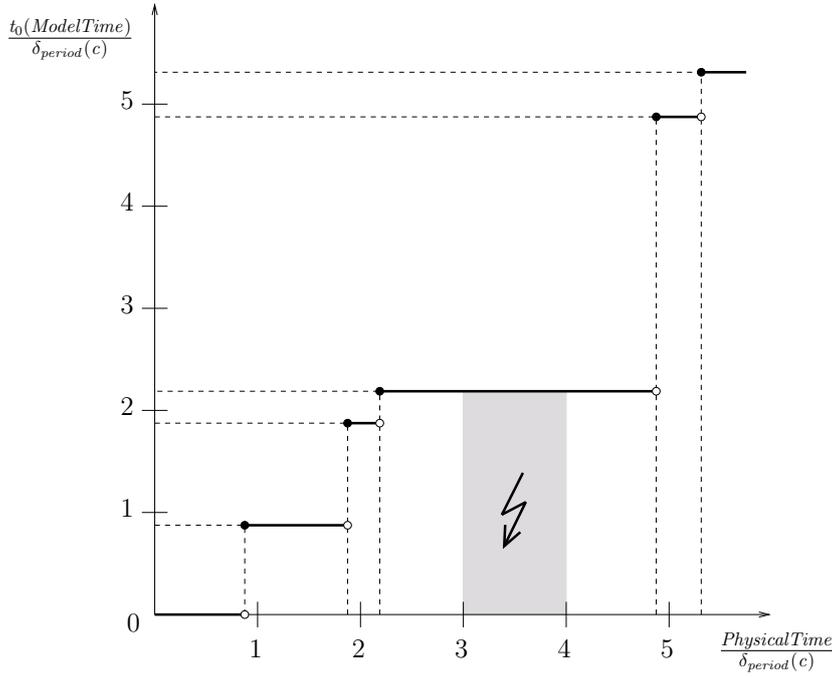


Figure 4.10: Model time synchronization scenario that fails in frame 3: at the beginning of frame 4, the model time is a time of frame 2 and therefore is too old.

the delay of flows, because we realize the *time-triggered* execution of flow calculations. See [Kop97] for a discussion about time-triggered vs. event-triggered systems.

In order to check time-triggeredness, we have a second constraint on states  $(c, v) \in S$ :

$$chktrig : S \rightarrow \mathbb{B}$$

$$(c, v) \mapsto nexttime_1(c, v) < physTime(v) \Rightarrow phase(sched(v)) = flow\_phase$$

where the function  $nexttime_1$  is defined as:

$$nexttime_1 : S \rightarrow \mathbb{R}_0^+$$

$$s \mapsto nexttime(1, s)$$

$$nexttime : \mathbb{N} \times S \rightarrow \mathbb{R}_0^+$$

$$(k, (c, v)) \mapsto \left( \left\lfloor \frac{t_0(modelTime(v))}{k \cdot \delta_{period}(c)} \right\rfloor + 1 \right) \cdot k \cdot \delta_{period}(c)$$

This calculates the physical time value  $t_{fixint}$  at which the subjects of the next flow phase shall be actually executed.  $t_{fixint}$  is the smallest multiple of the system period  $\delta_{period}(c)$  which is bigger than current model time, i.e. the beginning of the next frame.

The constraint  $chktrig$  then ensures that as soon as the time  $t_{fixint}$  is reached, a flow phase must be active. There is no need to ensure that this is no previous flow phase, because  $chktsync$  includes this.

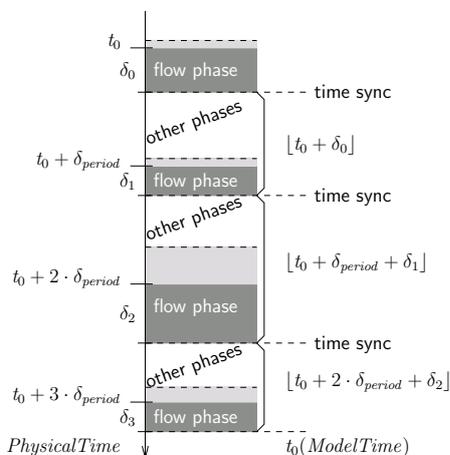


Figure 4.11: Successful execution: Model execution adjusts the model time  $modelTime(v)$  once for every system period  $\delta_{period}(c)$ .

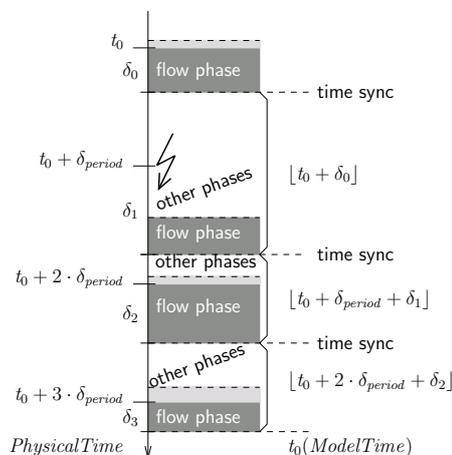


Figure 4.12: Timing failure: Flow execution is delayed.

Note that  $nexttime$  is the generalization of  $nexttime_1$ , which considers multiples of  $\delta_{period}(c)$ . It is used for the assignment of flows and interface modules to flow phases.  $nexttime$  is illustrated in figure 4.16.

### Synchronized and Fail States

Corresponding to the timing constraints given above, we have states  $S_{sync} \subset S$  within the state space that are synchronized, i.e.  $S_{sync} = \{s \in S \mid chktsync(s) \wedge chktrig(s)\}$ . The remaining states  $S_{fail} = S \setminus S_{sync}$  are *fail states*. If the system execution ever passes one of the fail states, the fail flag  $fail(v)$  will be set explicitly.

#### 4.4.2 Switching between Execution Phases

In this section, transitions  $t \in T$  of the state transition system are defined, which represent the switching of execution phases. Each of these transitions require that there is nothing more to do for any subject in the pre-state  $(c, v) \in S$ , i.e. that the schedule is empty and that the scheduler controls every LWP (which means that they are idle):

$$subj_{sched}(sched(v)) = \emptyset \wedge \forall p \in lwp(c) \bullet \kappa_{LWP}(v)(p) = scheduler$$

The switching of execution phases always takes some (small amount of) time  $\delta_{switch} \in \mathbb{R}_+$ .

**Switching to update\_phase.** The *update\_phase* is preceded by either a *notify\_phase*, a *flow\_phase*, or the *init\_phase*. When all scheduled transitions have been executed and their abstract machines are notified, or all active flows have been executed for one duration step, or all abstract subjects are initialized, respectively, the *update\_phase* is started. All abstract machines are scheduled

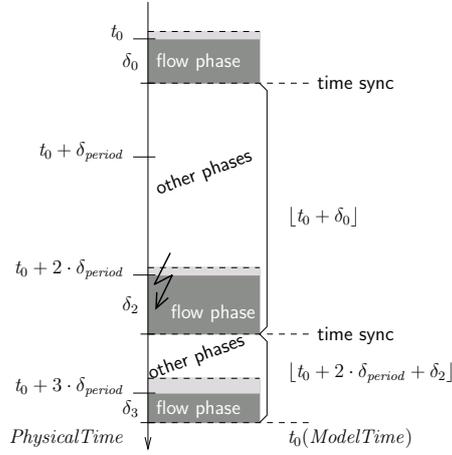


Figure 4.13: Timing failure: Non-flow phases take too long.

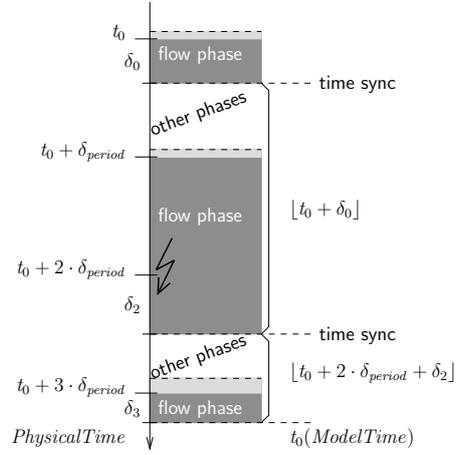


Figure 4.14: Timing failure: Flow phase takes too long.

to determine their enabled transitions and flows. Depending on the preceding phase, the model time is adjusted: (1) With a previous *notify\_phase*, the visible time remains constant, but only the causality tick is incremented. This provides a zero-time execution of transitions for the model. (2) After a *flow\_phase*, the visible time is synchronized with the physical time, such that for the model, time increases. (3) Succeeding the *init\_phase*, model time is not adjusted, i.e. it remains zero.

Let  $(c, v) \in S$  where for all LWPs  $p \in lwp(c)$  we have that  $\kappa_{LWP}(v)(p) = scheduler$ .

**Rule 4.4.1** If  $sched(v) = (\emptyset, notify\_phase)$ , then we have a transition  $(c, v) \longrightarrow (c, v')$  with  $v = v'$ , except for

$$\begin{aligned} sched(v') &= (am(c) \times \{update\}, update\_phase) \\ physTime(v') &= physTime(v) + \delta_{switch} \\ &\text{with } \delta_{switch} \in \mathbb{R}_+ \\ modelTime(v') &= t_0(modelTime(v)).(t_1(modelTime(v)) + 1) \quad \square \end{aligned}$$

**Rule 4.4.2** If  $sched(v) = (\emptyset, flow\_phase)$  and  $physTime(v) \geq nexttime_1(c, v)$ , then a transition  $(c, v) \longrightarrow (c, v')$  exists which has

$$\begin{aligned} sched(v') &= (am(c) \times \{update\}, update\_phase) \\ physTime(v') &= physTime(v) + \delta_{switch} \\ &\text{with } \delta_{switch} \in \mathbb{R}_+ \\ modelTime(v') &= \lfloor physTime(v) \rfloor.0 \end{aligned}$$

and otherwise  $v = v'$ . □

Note that flow phases cannot be left before the physical time for flow execution is reached. This ensures, that empty flow phases, i.e. flow phases which will

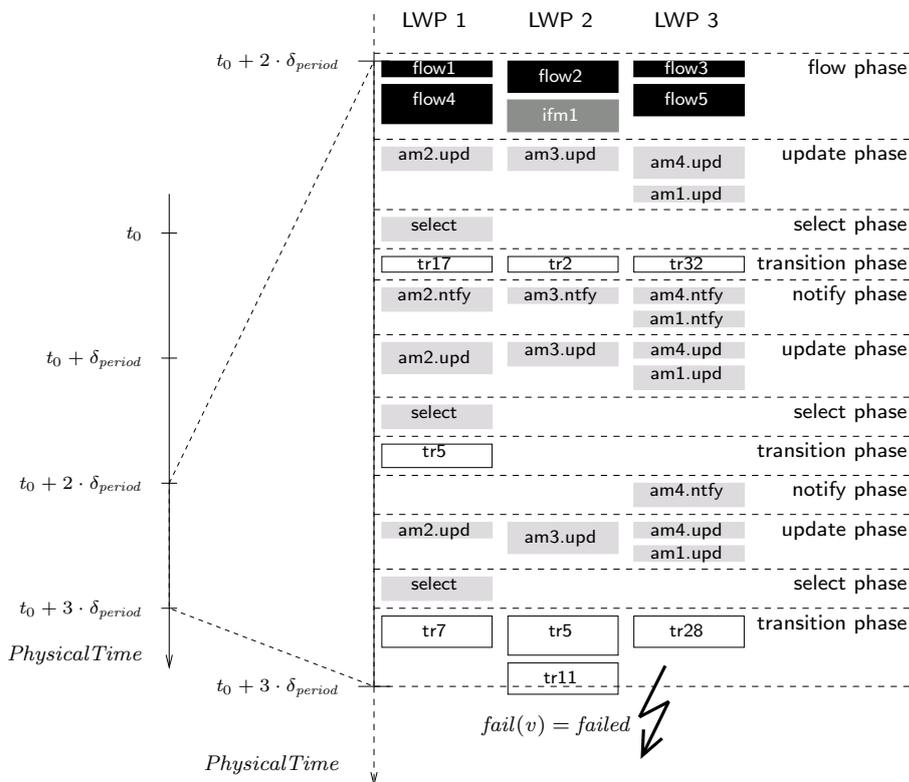


Figure 4.15: Scheduling example: timing failure.

neither schedule flows nor interface modules, are treated the same way as non-empty ones are.

Finally,

**Rule 4.4.3** if  $\text{sched}(v) = (\emptyset, \text{init\_phase})$ , then a transition  $(c, v) \longrightarrow (c, v')$  exists which modifies  $v$  only by

$$\begin{aligned} \text{sched}(v') &= (\text{am}(c) \times \{\text{update}\}, \text{update\_phase}) \\ \text{physTime}(v') &= \text{physTime}(v) + \delta_{\text{switch}} \\ &\text{with } \delta_{\text{switch}} \in \mathbb{R}_+ \end{aligned} \quad \square$$

**Switching to selection\_phase.** After an *update\_phase*, a *selection\_phase* follows. Control is taken over by the selector, in order to create a *selection* of discrete or flow steps from the abstract machine's states.

**Rule 4.4.4** Let  $(c, v) \in S$  where for all LWPs  $p \in \text{lwp}(c)$  we have that  $\kappa_{LWP}(v)(p) = \text{scheduler}$ , and moreover that  $\text{sched}(v) = (\emptyset, \text{update\_phase})$ .

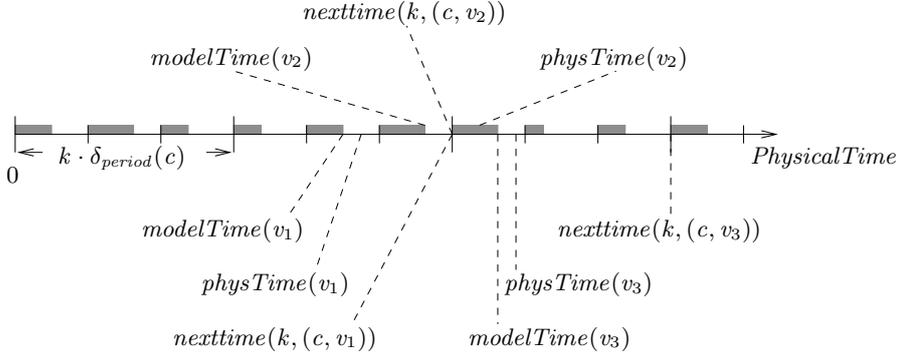


Figure 4.16: Illustration of the mapping  $nexttime$ , with factor  $k = 3$ : For every state  $(c, v) \in S$  and  $k$ , the beginning of the next frame such that  $fnotime(c, t)$  is a multiple of  $k$  is calculated. However, during the flow execution interval of that frame, its own start time results, as for  $(c, v_2)$  in this example.

We thus have a transition  $(c, v) \longrightarrow (c, v')$  with  $v = v'$ , except for

$$\begin{aligned} sched(v') &= (\{(selector, getSelection)\}, selection\_phase) \\ physTime(v') &= physTime(v) + \delta_{switch} \\ &\text{with } \delta_{switch} \in \mathbb{R}_+ \end{aligned} \quad \square$$

**Switching to transition\_phase or flow\_phase.** The transition to *transition\_phase* or *flow\_phase* depends on the selector's choice. Therefore it is defined with the selection rule in section 4.5.

**Switching to notify\_phase.** A *transition\_phase* is always succeeded by a *notify\_phase*. All abstract machines for which a transition was taken are scheduled to update their internal states accordingly.

**Rule 4.4.5** Let  $(c, v) \in S$  where for all LWPs  $p \in lwp(c)$  we have that  $\kappa_{LWP}(v)(p) = scheduler$ , and moreover that  $sched(v) = (\emptyset, transition\_phase)$ . We thus have a transition  $(c, v) \longrightarrow (c, v')$  with  $v = v'$ , except for

$$\begin{aligned} sched(v') &= (\{m \in am(c) \mid transAbsState(\kappa_{Subj_{abs}}(m)) \neq \lambda\} \\ &\quad \times \{notifyTrans\}, \\ &\quad notify\_phase) \\ physTime(v') &= physTime(v) + \delta_{switch} \\ &\text{with } \delta_{switch} \in \mathbb{R}_+ \end{aligned} \quad \square$$

### 4.4.3 Scheduling within Execution Phases

Within execution phases, single subjects have to be scheduled, depending on the schedule  $sched(v)$  of the current state  $(c, v) \in S$ , and depending on the availability of LWPs  $lwp(c)$ . Therefore, in this section transitions  $t \in T$  are defined that each represent the *allocation* of a single subject on an available

LWP. For the allocation of a subject, the corresponding LWP needs a time duration  $\delta_{alloc} \in \mathbb{R}_+$  before the execution of the subject's operation begins. We model this by adding  $\delta_{alloc}$  to the execution duration  $\delta_s \in \mathbb{R}_+$  of the operation or program, respectively.

The deallocation is not presented here, since it is a direct result of the termination of subject execution, and thus is included in the definitions of sections 4.5 and 4.6.

The way of how single subjects are allocated is different for abstract subjects and program subjects. Within each execution phase, only either kind of subjects can be executed, therefore we distinguish the allocation rules by these both kinds.

Further, there is a specific rule that lets time pass for idle LWPs. This is done exactly when a *flow\_phase* is active, but the next (physical time) frame is not reached yet. Thus, the system waits for the next flow execution interval.

**Dynamic LWP Assignment for Program Subjects.** Program subjects  $s \in \text{Subj}_{prog}$  in the schedule  $sched(v)$  are scheduled immediately on any idle LWP in order to execute operations  $op \in Op$ :

Let  $(c, v) \in S$  for some LWP  $p \in lwp(c)$ , such that  $\kappa_{LWP}(v)(p) = scheduler$ , and moreover that  $sched(v) = (sset, sphase)$  with  $(s, op) \in sset$ . Then subject  $s$  is selected and  $p$  is assigned to  $s$ , and  $s$  is immediately removed from the schedule. The program string is reset, i.e. the complete program string for operation  $op$  as defined in the HL<sup>3</sup> model is assigned, and a remaining duration for the first statement (including the duration needed for allocation) is set. The program's (optional) parameter is not modified.

If additionally,  $sphase = transition\_phase$  (which implies  $s \in Trans$ ), then the transition is set as the parameter for the next execution of the associated abstract machine, which will be the notification of the transition's execution. Internal state and duration are kept for the abstract machine.

**Rule 4.4.6** Thus, for  $sphase = transition\_phase$  we have a transition  $(c, v) \longrightarrow (c, v')$  with  $v = v'$ , except for

$$\begin{aligned}
\kappa_{LWP}(v') &= \kappa_{LWP}(v) \oplus \{p \mapsto s\} \\
sched(v') &= ((sset \setminus \{s\}), sphase) \\
\kappa_{Subj_{prog}}(v') &= \kappa_{Subj_{prog}}(v) \oplus \{(s, op) \mapsto \\
&\quad (prg_{subj}(c)(s, op), \delta_s + \delta_{alloc}, vis_{ProgState}(\kappa_{Subj_{prog}}(v)(s, op)))\} \\
&\quad \text{with } \delta_s, \delta_{alloc} \in \mathbb{R}_+ \\
\kappa_{Subj_{abs}}(v') &= \kappa_{Subj_{abs}}(v) \oplus \{am_{trans}(c)(s) \mapsto \\
&\quad (intState_{AbsState}(\kappa_{Subj_{abs}}(v)(am_{trans}(c)(s))), \\
&\quad \delta_{AbsState}(\kappa_{Subj_{abs}}(v)(am_{trans}(c)(s))), \\
&\quad s)\} \quad \square
\end{aligned}$$

**Rule 4.4.7** For all phases  $sphase \neq transition\_phase$ , we have a transition  $(c, v) \longrightarrow (c, v')$  that does not affect the execution state of abstract machines, i.e.  $v = v'$ , except for

$$\kappa_{LWP}(v') = \kappa_{LWP}(v) \oplus \{p \mapsto s\}$$

$$\begin{aligned}
sched(v') &= ((sset \setminus \{s\}), sphase) \\
\kappa_{Subjprog}(v') &= \kappa_{Subjprog}(v) \oplus \{(s, op) \mapsto \\
&\quad (prg_{subj}(c)(s, op), \delta_s + \delta_{alloc}, vis_{ProgState}(\kappa_{Subjprog}(v)(s, op)))\} \\
&\quad \text{with } \delta_s, \delta_{alloc} \in \mathbb{R}_+ \quad \square
\end{aligned}$$

There is an additional restriction for  $sphase = flow\_phase$  – the above transition is only allowed, if  $physTime(v) \geq nexttime_1(c, v)$ . That means, within a flow phase, the subjects in the schedule are scheduled as recently as the physical time for the next flow executions has come. Previously, physical time just has to pass.

**Waiting for Flow Execution.** A flow phase is idle and lets time pass, as long as the physical time for the next flow executions wrt. the system period is not reached yet.

**Rule 4.4.8** Let  $(c, v) \in S$  with  $phase(sched(v)) = flow\_phase$  and  $physTime(v) < nexttime_1(c, v)$ . Then some time  $\delta \in \mathbb{R}_+$  passes, i.e. transitions  $(c, v) \longrightarrow (c, v') \in T$  exist with  $v = v'$ , but

$$\begin{aligned}
physTime(v') &= physTime(v) + \delta \\
&\quad \text{with } \delta \leq nexttime_1(c, v) - physTime(v) \quad \square
\end{aligned}$$

**Static LWP Assignment for Abstract Subjects.** An abstract subject  $s \in Subj_{abs}$  in the schedule  $sched(v)$  is scheduled for operation  $op \in Op$  on its statically assigned LWP  $p$ , as soon as it is available:

**Rule 4.4.9** Let  $(c, v) \in S$  for  $sched(v) = (sset, sphase)$  with  $(s, op) \in sset$ , and let  $\kappa_{LWP}(v)(lwp_{subj}(c)(s)) = scheduler$ . Then  $p$  is mapped to subject  $s$ , and  $s$  is removed from the schedule. The internal state and the subject's (optional) parameter are retained, but a remaining duration for the allocation and execution of the operation is set. We have a transition  $(c, v) \longrightarrow (c, v')$  with  $v = v'$ , except for

$$\begin{aligned}
\kappa_{LWP}(v') &= \kappa_{LWP}(v) \oplus \{lwp_{subj}(c)(s) \mapsto s\} \\
sched(v') &= ((sset \setminus \{s\}), sphase) \\
\kappa_{Subj_{abs}}(v') &= \kappa_{Subj_{abs}}(v) \oplus \{s \mapsto (intState_{AbsState}(\kappa_{Subj_{abs}}(v)(s)), \\
&\quad \delta_s + \delta_{alloc}, trans_{AbsState}(\kappa_{Subj_{abs}}(v)(s)))\} \\
&\quad \text{with } \delta_s, \delta_{alloc} \in \mathbb{R}_+ \quad \square
\end{aligned}$$

## 4.5 Abstract Subject Execution

In this section, the execution of abstract subjects is defined by transitions  $t \in T$ . Abstract subjects are executed in that a specific operation of the corresponding subject is run. The execution of the operation takes some time before it is terminated. The operation's effect on the state space becomes effective with its termination.

Further, for  $|lwp(c)| > 1$  it is possible that several abstract subjects are executed in parallel, i.e. time evolves for all active subjects in common. This is defined for all possible abstract subject operations by the *progress rule* given below.

As a side effect of progress, the timing constraints identified in section 4.4.1 may be violated. This is regarded in the progress rule, and leads to the setting of the fail flag.

In contrast to the *progress* of operations, the *effect* of operations depends on the specific operation and the corresponding kind of subject. Therefore, different *termination rules* are provided for the possible operations, i.e. for

$$op \in \{(Am, init), (Am, update), (Am, notifyTrans), (selector, init), (selector, getSelection)\}$$

### 4.5.1 Progress of Abstract Subject Execution

Operations of abstract subjects are executed, as soon as a maximum subset from the schedule is allocated on LWPs, such that there is no idle LWP left or the schedule is empty. Then, operations take some time.

Let  $(c, v) \in S$  with  $phase(sched(v)) \in \{update\_phase, selection\_phase, notify\_phase\}$ . Given that all active subjects have a positive remaining execution duration, i.e.  $\forall s \in \text{ran } \kappa_{LWP}(v) \bullet \delta_{AbsState}(\kappa_{Subj_{abs}}(v)(s)) \in \mathbb{R}_+$ , and that there are no more idle LWPs left that could be used to execute some subject in the schedule, i.e.  $\neg \exists s \in subj_{sched}(sched(v)) \bullet \kappa_{LWP}(v)(lwp_{subj}(c)(s)) = scheduler$ . Then arbitrary time durations can elapse, as long as the first subject is not completed, yet.

Consider the potential successor states  $Suc \subseteq S$ , which are given for all  $\delta \leq \delta_{min}$ , whereas  $\delta_{min}$  is the minimum value of the set of all remaining durations  $\{\delta \in \mathbb{R}_+ \mid \exists s \in \text{dom } \kappa_{Subj_{abs}}(v) \bullet \delta = \delta_{AbsState}(\kappa_{Subj_{abs}}(v)(s))\}$ , such that  $v = v'$  holds for  $(c, v') \in Suc$ , except for

$$\begin{aligned} \kappa_{Subj_{abs}}(v') &= \kappa_{Subj_{abs}}(v) \oplus \{s \mapsto \\ &\quad (intState_{AbsState}(\kappa_{Subj_{abs}}(v)(s)), \delta_{AbsState}(\kappa_{Subj_{abs}}(v)(s)) - \delta, \\ &\quad trans_{AbsState}(\kappa_{Subj_{abs}}(v)(s)) \\ &\quad \mid s \in \text{dom } \kappa_{Subj_{abs}}(v) \wedge \delta_{AbsState}(\kappa_{Subj_{abs}}(v)(s)) \in \mathbb{R}_+\} \\ physTime(v') &= physTime(v) + \delta \end{aligned}$$

Since between  $(c, v)$  and  $(c, v')$  physical time elapses, time synchronization (see section 4.4.1) is monitored here. The successor states are therefore divided into  $Suc_{sync} = Suc \cap S_{sync}$  and  $Suc_{fail} = Suc \cap S_{fail}$ .

**Rule 4.5.1** For all  $(c, v') \in Suc_{sync}$ , there are transitions  $(c, v) \longrightarrow (c, v') \in T$  that let the respective time durations  $\delta \leq \delta_{min}$  pass.  $\square$

**Rule 4.5.2** For the states  $(c, v') \in Suc_{fail}$  we have transitions  $(c, v) \longrightarrow (c, v'') \in T$  that represent the evolution of time durations in the same way, but also set the fail flag. Therefore,  $v' = v''$  holds, with the exception of

$$fail(v'') = failed \quad \square$$

Note that timing failures are thus detected in finite time, which happens with at most the delay of  $\delta_{min}$ .

### 4.5.2 Termination of Abstract Subject Execution

**Termination of Abstract Machines' Initialization.** Initialization of abstract machines assigns a pre-defined internal state to the corresponding abstract machine, provided by the mapping

$$init_{Am} : Am \rightarrow IntState$$

**Rule 4.5.3** Then, for states  $(c, v) \in S$  with an abstract machine  $m$  for which the current execution time has elapsed in the *init\_phase*, that is  $\delta_{AbsState}(\kappa_{Subj_{abs}}(v)(m)) = 0$  and  $phase(sched(v)) = init\_phase$ , there is a transition  $(c, v) \longrightarrow (c, v') \in T$  with

$$\begin{aligned} \kappa_{Subj_{abs}}(v') &= \kappa_{Subj_{abs}}(v) \oplus \{m \mapsto (init_{Am}(m), \infty, \lambda)\} \\ \kappa_{LWP}(v') &= \kappa_{LWP}(v) \oplus \{p \mapsto scheduler \mid \kappa_{LWP}(v)(p) = m\} \end{aligned} \quad \square$$

such that the internal state is assigned, and the formerly used LWP is released.

**Termination of Abstract Machines' Update.** The effect of the operation *update* is that the abstract machine (1) has an actual internal state, (2) has an actual abstract machine state, and (3) each of the flows it controls is enabled or disabled, corresponding to the actual internal state. Input for the *update* operation is the previous internal state of the abstract machine, along with the parts of the state space that are visible to subjects, including the current channel valuations.

Formally, the effect is given by a mapping from the internal abstract machine state and the model-accessible portion of the state space  $S$  to a succeeding abstract machine state, along with a flow enabling function:

$$\begin{aligned} UpdState_{pre} &= Am \times IntState \times CONST_m \times VAR_{mread} \\ UpdState_{post} &= IntState \times AmState \times (Flow \leftrightarrow \mathbb{B}) \\ update &: UpdState_{pre} \rightarrow UpdState_{post} \end{aligned}$$

The resulting components are accessible through

$$\begin{aligned} update_{IntState} &: UpdState_{pre} \rightarrow IntState \\ s &\mapsto \pi_1 UpdState_{post}(update(s)) \\ update_{AmState} &: UpdState_{pre} \rightarrow AmState \\ s &\mapsto \pi_2 UpdState_{post}(update(s)) \\ update_{Flow} &: UpdState_{pre} \rightarrow (Flow \leftrightarrow \mathbb{B}) \\ s &\mapsto \pi_3 UpdState_{post}(update(s)) \end{aligned}$$

**Rule 4.5.4** For states  $(c, v) \in S$  which have an abstract machine  $m \in am(c)$  whose current operation is about to terminate within the update phase, i.e.  $\delta_{AbsState}(\kappa_{Subj_{abs}}(v)(m)) = 0$  and  $phase(sched(v)) = update\_phase$ , we have a transition  $(c, v) \longrightarrow (c, v') \in T$  with

$$\kappa_{Subj_{abs}}(v') = \kappa_{Subj_{abs}}(v) \oplus \{m \mapsto (update_{IntState}(preState), \infty, \lambda)\}$$

$$\begin{aligned}
\kappa_{LWP}(v') &= \kappa_{LWP}(v) \oplus \{p \mapsto \text{scheduler} \mid \kappa_{LWP}(v)(p) = m\} \\
\kappa_{Am}(v') &= \kappa_{Am}(v) \oplus \{m \mapsto \text{update}_{AmState}(\text{preState})\} \\
\kappa_{Flow}(v') &= \kappa_{Flow}(v) \oplus \\
&\quad (\{f \mapsto \infty \mid f \in \text{dom}(\text{update}_{Flow}(\text{preState})) \\
&\quad \quad \wedge \neg \text{update}_{Flow}(\text{preState})(f) \wedge \text{am}_{flow}(c)(f) = m\} \\
&\quad \cup \\
&\quad \{f \mapsto \text{nexttime}_{flow}(f, (c, v)) \mid f \in \text{dom}(\text{update}_{Flow}(\text{preState})) \\
&\quad \quad \wedge \text{update}_{Flow}(\text{preState})(f) \wedge \text{am}_{flow}(c)(f) = m\})
\end{aligned}$$

where

$$\begin{aligned}
\text{preState} &= (m, \text{intState}_{AbsState}(\kappa_{Subj_{abs}}(v)(m)), \\
&\quad (\text{subject}_{var}(c), \text{chan}_{port}(c), \text{subject}_{port}(c)), \\
&\quad (\text{modelTime}(v), \sigma_{var}(v), \kappa_{Chan}(v))) \quad \square
\end{aligned}$$

Here, (1) the internal state of  $m$  is updated, and the remaining execution duration is reset. (2)  $m$  is deallocated from its current LWP. (3) The abstract machine state is updated. (4) All flows under control of  $m$  are either disabled, or get their new execution time. Here a new value is assigned to indicate when flows are to be scheduled. A flow is supposed to get scheduled at the next instance where its period would allow its execution:

$$\begin{aligned}
\text{nexttime}_{flow} : Flow \times S &\rightarrow \mathbb{R}_0^+ \\
(f, (c, v)) &\mapsto \text{nexttime}(\text{period}_{flow}(c)(f), (c, v))
\end{aligned}$$

**Termination of Abstract Machines' Notification.** The effect of the operation *notifyTrans* is that the abstract machine has an updated internal state, which regards the previous execution of one of its controlled transitions. Input for the *notifyTrans* operation is the previous internal state of the abstract machine, along with the executed transition.

Formally, the effect is given by a mapping from the internal abstract machine state and the executed transition  $t$  to a succeeding internal abstract machine state:

$$\text{notify} : Am \times \text{IntState} \times \text{Trans} \rightarrow \text{IntState}$$

**Rule 4.5.5** For states  $(c, v) \in S$  which have an abstract machine  $m \in \text{am}(c)$  whose current operation is about to terminate within the notify phase, i.e.  $\delta_{AbsState}(\kappa_{Subj_{abs}}(v)(m)) = 0$  and  $\text{phase}(\text{sched}(v)) = \text{notify\_phase}$ , we have a transition  $(c, v) \longrightarrow (c, v') \in T$  with

$$\begin{aligned}
\kappa_{Subj_{abs}}(v') &= \kappa_{Subj_{abs}}(v) \oplus \{m \mapsto (\text{notify}(m, \\
&\quad \text{intState}_{AbsState}(\kappa_{Subj_{abs}}(v)(m)), \\
&\quad \text{trans}_{AbsState}(\kappa_{Subj_{abs}}(v)(m)), \\
&\quad \infty, \lambda)\} \\
\kappa_{LWP}(v') &= \kappa_{LWP}(v) \oplus \{p \mapsto \text{scheduler} \mid \kappa_{LWP}(v)(p) = m\} \quad \square
\end{aligned}$$

The execution state of the abstract machine is modified, such that (1) the internal state is updated, (2) the remaining execution duration is reset, (3) and the parameter entry, which was the executed transition, is removed. The corresponding LWP is released, i.e. it is controlled by the scheduler now.

**Termination of Selector's Initialization.** The initialization of the selector assigns a pre-defined internal state to it, and further determines if the state space is *well-formed*, corresponding to constraints which depend on the selector's high-level formalism. This is given by the mapping

$$init_{sel} : S \rightarrow IntState \times \mathbb{B}$$

and the corresponding projections

$$intState_{init_{sel}} : S \rightarrow IntState$$

$$s \mapsto \pi_1 init_{sel}(s)$$

$$wellFormed_{init_{sel}} : S \rightarrow \mathbb{B}$$

$$s \mapsto \pi_2 init_{sel}(s)$$

**Rule 4.5.6** Then, for states  $(c, v) \in S$  for which the selector's execution time has elapsed in the *init\_phase*, that is  $\delta_{AbsState}(\kappa_{Subj_{abs}}(v)(selector)) = 0$  and  $phase(sched(v)) = init\_phase$ , there is a transition  $(c, v) \longrightarrow (c, v') \in T$  with

$$\begin{aligned} \kappa_{Subj_{abs}}(v') &= \kappa_{Subj_{abs}}(v) \oplus \{selector \mapsto (intState_{init_{sel}}(c, v), \infty, \lambda)\} \\ \kappa_{LWP}(v') &= \kappa_{LWP}(v) \oplus \{p \mapsto scheduler \mid \kappa_{LWP}(v)(p) = selector\} \\ fail(v') &= \begin{cases} ok & \text{if } wellFormed_{init_{sel}}(c, v) \\ failed & \text{else} \end{cases} \quad \square \end{aligned}$$

such that the internal state is assigned, the formerly used LWP is released, and the fail flag is set according to the initialization's result.

**Termination of Selector's Selection.** The actual behavior of the *selector* component depends on both the high-level formalism (and the respective transformation function to HL<sup>3</sup>) and a user-defined *selection policy* determined by the context which the real-time execution is investigated in. The *selector's* behavior thus is treated as an abstract subject which takes time for its execution and provides a *selection* upon termination. The selection is mainly determined from the states  $\kappa_{Am}(v)$  of the abstract machines, along with its internal state. Additionally, the model time as well as a dedicated set of ports may be accessed.

$$Selection = \mathbb{B} \times (Trans \leftrightarrow VisibilitySet)$$

$$select : ModelTime \times \mathcal{P}(Port) \times IntState \times \Sigma_{Am}$$

$$\rightarrow IntState \times Selection \times ChanState$$

The selection consists of (1) a boolean value denoting whether a flow step is possible and (2) a set of transitions that constitute a possible discrete step. The set of transitions is given by the domain of a mapping, which also provides a visibility set for each of the transitions. The time stamps at which the results of the transitions' actions become effective for the respective recipients therefore

depend on the high-level formalism. (3) Further, a new internal state results for the selector. (4) As a side-effect, data can be written to channels.

The components of the selection are directly given by

$$\begin{aligned} sel_{flow} &: ModelTime \times \mathcal{P}(Port) \times IntState \times \Sigma_{Am} \rightarrow \mathbb{B} \\ &(tick, P, s_i, s_{am}) \mapsto \pi_1(\pi_2 select(tick, P, s_i, s_{am})) \\ sel_{trans, vis} &: ModelTime \times \mathcal{P}(Port) \times IntState \times \Sigma_{Am} \\ &\rightarrow (Trans \leftrightarrow VisibilitySet) \\ &(tick, P, s_i, s_{am}) \mapsto \pi_2(\pi_2 select(tick, P, s_i, s_{am})) \end{aligned}$$

Transitions without visibility sets are directly accessible by

$$\begin{aligned} sel_{trans} &: ModelTime \times \mathcal{P}(Port) \times IntState \times \Sigma_{Am} \rightarrow \mathcal{P}(Trans) \\ &(tick, P, s_i, s_{am}) \mapsto \text{dom } sel_{trans, vis}(tick, P, s_i, s_{am}) \end{aligned}$$

The new internal state can be accessed through

$$\begin{aligned} sel_{intState} &: ModelTime \times \mathcal{P}(Port) \times IntState \times \Sigma_{Am} \rightarrow IntState \\ &(tick, P, s_i, s_{am}) \mapsto \pi_1 select(tick, P, s_i, s_{am}) \end{aligned}$$

The resulting channel state is given by

$$\begin{aligned} sel_{chanEntry} &: ModelTime \times \mathcal{P}(Port) \times IntState \times \Sigma_{Am} \rightarrow ChanState \\ &(tick, P, s_i, s_{am}) \mapsto \pi_3 select(tick, P, s_i, s_{am}) \end{aligned}$$

There are several constraints that must hold for the results of the *select* effect: (1) The set of selected transitions is chosen according to the sets of enabled transitions of the abstract machines. (2) In this set there is at most one associated transition for each abstract machine. (3) The selector is only allowed to allow a flow step, if all abstract machines agree. (4) The selector is not allowed to mischievously block the execution of an HL<sup>3</sup> program – in case that no flow is possible, but some abstract machines do have transitions enabled, at least one transition will be selected.

$$\begin{aligned} &\forall (tick, P, s_i, s_{am}) \in \text{dom } select \bullet \\ &sel_{trans}(tick, P, s_i, s_{am}) \subseteq \bigcup_{m \in \text{dom } s_{am}} trans_{AmState}(s_{am}(m)) \\ &\forall (tick, P, s_i, s_{am}) \in \text{dom } select \bullet \\ &\forall t_1, t_2 \in sel_{trans}(tick, P, s_i, s_{am}) \bullet \exists m_1, m_2 \in \text{dom } s_{am} \bullet \\ & \quad t_1 \in trans_{AmState}(s_{am}(m_1)) \wedge t_2 \in trans_{AmState}(s_{am}(m_2)) \\ & \quad \wedge (t_1 \neq t_2 \Rightarrow m_1 \neq m_2) \\ &\forall (tick, P, s_i, s_{am}) \in \text{dom } select \bullet \\ & \quad sel_{flow}(tick, P, s_i, s_{am}) \Rightarrow \forall m \in \text{dom } s_{am} \bullet flow_{AmState}(s_{am}(m)) \\ &\forall (tick, P, s_i, s_{am}) \in \text{dom } select \bullet \\ & \quad \left( \bigcup_{m \in \text{dom } s_{am}} trans_{AmState}(s_{am}(m)) \neq \emptyset \wedge \neg sel_{flow}(s_{am}) \right) \\ & \quad \Rightarrow sel_{trans}(s) \neq \emptyset \end{aligned}$$

Let  $(c, v) \in S$  where within the selection phase, the selector's current operation is about to terminate, i.e.  $sched(v) = (\emptyset, selection\_phase)$  and  $\delta_{AbsState}(\kappa_{Subj_{abs}}(v)(selector)) = 0$ . Further, let the current internal state of the selector  $s_i = intState_{AbsState}(\kappa_{Subj_{abs}}(v)(selector))$ . Then, the selector's operation  $getSelection$  is finished, and provides a selection.

**Rule 4.5.7** If this selection contains any transitions, that means if  $sel_{trans}(s_i, \kappa_{Am}(v)) \neq \emptyset$ , we have a transition  $t_{disc} = (c, v) \longrightarrow (c, v')$  with  $v = v'$ , except that the transitions of the selection constitute the scheduled subjects now, and further the visibility sets are set as parameters for the transitions' execution states, in combination with the transitions' static visibility sets (see section 4.3 for operation  $\boxplus$ ):

$$\begin{aligned}
sched(v') &= (sel_{trans}(modelTime(v), selport(c), s_i, \kappa_{Am}(v)) \times \{action\}, \\
&\quad transition\_phase) \\
physTime(v') &= physTime(v) + \delta_{switch} \\
&\quad \text{with } \delta_{switch} \in \mathbb{R}_+ \\
\kappa_{Subj_{prog}}(v') &= \kappa_{Subj_{prog}}(v) \oplus \\
&\quad \{(t, action) \mapsto (stringProgState(\kappa_{Subj_{prog}}(v)(t, action)), \\
&\quad \delta_{ProgState}(\kappa_{Subj_{prog}}(v)(t, action)), vis_{trans}(c)(t) \boxplus vis) \\
&\quad \mid (t \mapsto vis) \in sel_{trans, vis}(modelTime(v), selport(c), s_i, \kappa_{Am}(v))\} \\
\kappa_{Subj_{abs}}(v') &= \kappa_{Subj_{abs}}(v) \oplus \\
&\quad \{selector \mapsto (sel_{intState}(modelTime(v), selport(c), s_i, \kappa_{Am}(v)), \infty, \lambda)\} \\
\kappa_{Chan}(v') &= \kappa_{Chan}(v) \oplus \\
&\quad \{insert_{Port}(p, (data, \{tick, p\}), \\
&\quad \quad (subject_{var}(c), chan_{port}(c), subject_{port}(c)), \\
&\quad \quad (modelTime(v), \sigma_{var}(v), \kappa_{Chan}(v)))\} \\
&\quad \mid (tick, p) \mapsto data \\
&\quad \in sel_{chanEntry}(modelTime(v), selport(c), s_i, \kappa_{Am}(v))\} \quad \square
\end{aligned}$$

Additionally, the selector's internal state is updated, and the data values are inserted into the respective channels.

**Rule 4.5.8** We have another transition  $t_{flow} = (c, v) \longrightarrow (c, v'')$  provided that  $sel_{flow}(s_i, \kappa_{Am}(v))$ , i.e. that the selection allows a flow step.

$$\begin{aligned}
sched(v'') &= \\
&\quad ((\{f \in flow(c) \mid \kappa_{Flow}(v)(f) = nexttime_1(c, v)\} \times \{integrate\}) \\
&\quad \cup \{\{md \in ifm(c) \mid nexttime_{ifm, poll}(md, (c, v)) = \\
&\quad \quad nexttime_1(c, v)\} \times \{poll\}\}) \\
&\quad \cup \{\{md \in ifm(c) \mid nexttime_{ifm, tmit}(md, (c, v)) = \\
&\quad \quad nexttime_1(c, v)\} \times \{transmit\}\}) \\
&\quad , flow\_phase) \\
physTime(v'') &= physTime(v) + \delta_{switch} \\
&\quad \text{with } \delta_{switch} \in \mathbb{R}_+ \\
\kappa_{Subj_{prog}}(v'') &= \kappa_{Subj_{prog}}(v) \oplus
\end{aligned}$$

$$\begin{aligned}
& \{(f, \text{integrate}) \mapsto (\text{string}_{\text{ProgState}}(\kappa_{\text{Subj}_{\text{prog}}}(v))(f, \text{integrate})), \\
& \quad \delta_{\text{ProgState}}(\kappa_{\text{Subj}_{\text{prog}}}(v))(f, \text{integrate}), \text{vis}_{\text{flow}}(c)(f) \uplus (\text{port}(c) \times \{0.1\}) \\
& \quad | f \in \text{flow}(c) \bullet \kappa_{\text{Flow}}(v)(f) = \text{nexttime}_1(c, v)\} \\
& \cup \{(md, \text{poll}) \mapsto (\text{string}_{\text{ProgState}}(\kappa_{\text{Subj}_{\text{prog}}}(v))(md, \text{poll})), \\
& \quad \delta_{\text{ProgState}}(\kappa_{\text{Subj}_{\text{prog}}}(v))(md, \text{poll}), \text{vis}_{\text{ifm}}(c)(md) \uplus (\text{port}(c) \times \{0.1\}) \\
& \quad | md \in \text{ifm}(c) \bullet \text{nexttime}_{\text{ifm}, \text{poll}}(md, (c, v)) \\
& \quad \quad = \text{nexttime}_1(c, v)\} \\
& \kappa_{\text{Subj}_{\text{abs}}}(v') = \kappa_{\text{Subj}_{\text{abs}}}(v) \oplus \\
& \quad \{\text{selector} \mapsto (\text{sel}_{\text{intState}}(\text{modelTime}(v), \text{selport}(c), s_i, \kappa_{\text{Am}}(v)), \infty, \lambda)\} \\
& \kappa_{\text{Chan}}(v') = \kappa_{\text{Chan}}(v) \oplus \\
& \quad \{\text{insert}_{\text{Port}}(p, (\text{data}, \{\text{tick}, p\})), \\
& \quad \quad (\text{subject}_{\text{var}}(c), \text{chan}_{\text{port}}(c), \text{subject}_{\text{port}}(c)), \\
& \quad \quad (\text{modelTime}(v), \sigma_{\text{Var}}(v), \kappa_{\text{Chan}}(v)) \\
& \quad | (\text{tick}, p) \mapsto \text{data} \\
& \quad \quad \in \text{sel}_{\text{chanEntry}}(\text{modelTime}(v), \text{selport}(c), s_i, \kappa_{\text{Am}}(v))\} \quad \square
\end{aligned}$$

Here, (1) flows and interface modules are set to the schedule. For interface modules, the fixed periods determine which of them are scheduled, whereas for flows, the flows' execution states denote this. (2) Additionally, visibility sets are generated from the static visibility sets of flows and interface modules, such that the calculation results will be published in the model future. The visibility sets are set as parameters to the flows' and interface modules' execution states. (3) The selector's new internal state is inserted. (4) Data is written to the corresponding channels.

The calculation of the scheduling time ticks for the polling of interface modules as well as the transmission of data to them is defined in the same fashion as the calculation of the scheduling time ticks of flows:

$$\begin{aligned}
& \text{nexttime}_{\text{ifm}, \text{poll}} : \text{Ifm} \times S \rightarrow \mathbb{R}_0^+ \\
& \quad (md, (c, v)) \mapsto \text{nexttime}(\text{period}_{\text{ifm}, \text{poll}}(c)(f), (c, v)) \\
& \text{nexttime}_{\text{ifm}, \text{tmit}} : \text{Ifm} \times S \rightarrow \mathbb{R}_0^+ \\
& \quad (md, (c, v)) \mapsto \text{nexttime}(\text{period}_{\text{ifm}, \text{tmit}}(c)(f), (c, v))
\end{aligned}$$

Note that (1) one of transitions  $\{t_{\text{disc}}, t_{\text{flow}}\}$  is non-deterministically chosen, if  $\text{sel}_{\text{trans}}(s_i, \kappa_{\text{Am}}(v)) \neq \emptyset \wedge \text{sel}_{\text{flow}}(s_i, \kappa_{\text{Am}}(v))$ , and that (2) a deadlock occurs, if  $\text{sel}_{\text{trans}}(s_i, \kappa_{\text{Am}}(v)) = \emptyset \wedge \neg \text{sel}_{\text{flow}}(s_i, \kappa_{\text{Am}}(v))$ . A well-formed model of the applied high-level formalism should avoid the second situation.

## 4.6 Program Subject Execution

This section defines the behavior of program subjects by transitions  $t \in T$ . Program subjects are similar to abstract subjects in that they have operations which can be executed, which takes some time before the execution terminates.

In contrast to abstract subjects, the operations of program subjects are given in a more concrete way – by *programs*. Such a program is a *sequence of*

*statements*, therefore the execution of an operation is given by the sequence of executions of these statements.

Each statement execution then has its own duration, and its own effect on the state space on termination of the statement. Therefore, we define the progress of parallel statement executions by one *progress rule* which applies to any set of statements. According to abstract subjects, for  $|lwp(c)| > 1$  time evolves *in common* for several operations and therefore for several statements. Here, timing constraints (see section 4.4.1) may be violated, too, leading to the setting of the fail flag.

The *effects* are defined per statement. We provide the corresponding *termination rule* in the following. The termination of a complete program is defined for the empty program string, which results when the last statement in a program is terminated, as well as for an explicit `return` statement, separately.

### 4.6.1 Progress of Statement Execution

Statements of program subjects are executed, as soon as a maximum subset from the schedule is allocated on LWPs, such that there is no idle LWP left or the schedule is empty. Then, statements take some time.

Let  $(c, v) \in S$  with  $phase(sched(v)) \in \{flow\_phase, transition\_phase\}$ . Suppose that all active subjects have a positive remaining execution duration, i.e.  $\forall s \in \text{ran } \kappa_{LWP}(v) \bullet \delta_{ProgState}(\kappa_{Subjprog}(v)(s)) \in \mathbb{R}_+$ , and that a maximum subset of program subjects from the schedule is allocated on LWPs, such that there is no idle LWP left or the schedule is empty. That is,  $scheduler \notin \text{ran } \kappa_{LWP}(v) \vee subj_{sched}(sched(v)) = \emptyset$ . Then some time can pass for the currently active statements to complete.

**Rule 4.6.1** Let  $\delta_{min}$  denote the minimal value of all remaining execution times of active statements captured by  $\{\delta \in \mathbb{R}_+ \mid \exists (s, op) \in \text{dom } \kappa_{Subjprog}(v) \bullet \delta = \delta_{ProgState}(\kappa_{Subjprog}(v)(s, op))\}$ . Then for all  $\delta \leq \delta_{min}$ , states  $(c, v') \in Suc \subseteq S$  exist with the difference between  $v$  and  $v'$  of

$$\begin{aligned} \kappa_{Subjprog}(v') &= \kappa_{Subjprog}(v) \oplus \{ (s, op) \mapsto \\ &\quad (string_{ProgState}(\kappa_{Subjprog}(v)(s, op)), \delta_{ProgState}(\kappa_{Subjprog}(v)(s, op)) - \delta \\ &\quad \vee is_{ProgState}(\kappa_{Subjprog}(v)(s, op))) \mid (s, op) \in \text{dom } \kappa_{Subjprog}(v) \\ &\quad \wedge \delta_{ProgState}(\kappa_{Subjprog}(v)(s, op)) \in \mathbb{R}_+ \} \\ physTime(v') &= physTime(v) + \delta \end{aligned} \quad \square$$

Similarly to the progress of abstract subject operations, the progress of statements may cause a timing failure. Therefore, the successor states are partitioned into  $Suc_{sync} = Suc \cap S_{sync}$  and  $Suc_{fail} = Suc \cap S_{fail}$ .

**Rule 4.6.2** Transitions  $(c, v) \longrightarrow (c, v') \in T$  exist for states  $(c, v') \in Suc_{sync}$ , whereas for  $(c, v') \in Suc_{fail}$  we have transitions  $(c, v) \longrightarrow (c, v'') \in T$ , such that the fail flag is set:  $v' = v''$  holds, except for

$$fail(v'') = failed \quad \square$$

### 4.6.2 Termination of Statement Execution

Suppose that in state  $(c, v) \in S$  a program statement  $stmt \in Stmt \setminus \{\mathbf{return}\}$  is about to terminate, i.e. there is an operation  $(s, op) \in \text{dom } \kappa_{Subj_{prog}}(v)$  with

$$\kappa_{Subj_{prog}}(v)(s, op) = (\langle stmt \rangle \hat{\ } prg, 0, p_{vis})$$

Then the termination of  $stmt$  has an effect on the state space, given by the effect function

$$\begin{aligned} \epsilon : Program \times Param_{prog} \times Subject \times CONST_m \times VAR_{mread} \\ \rightarrow Program \times VAR_{mwrite} \end{aligned}$$

From a given program, a parameter, as well as the part of the state space that is readable from HL<sup>3</sup> models, a remaining program along with the (potentially) modified state space portion with write access from the model results. The effect's resulting components are given by

$$\begin{aligned} \epsilon_{prg} : Program \times Param_{prog} \times Subject \times CONST_m \times VAR_{mread} &\rightarrow Program \\ (prg, param, s, c, v) &\mapsto \pi_1 \epsilon(prg, param, c, v) \\ \epsilon_{var} : Program \times Param_{prog} \times Subject \times CONST_m \times VAR_{mread} &\rightarrow \Sigma_{Var} \\ (prg, param, s, c, v) &\mapsto \pi_1 (\pi_2 \epsilon(prg, param, c, v)) \\ \epsilon_{chan} : Program \times Param_{prog} \times Subject \times CONST_m \times VAR_{mread} &\rightarrow \Sigma_{Chan} \\ (prg, param, s, c, v) &\mapsto \pi_2 (\pi_2 \epsilon(prg, param, c, v)) \end{aligned}$$

**Rule 4.6.3** From this, a transition  $(c, v) \longrightarrow (c, v') \in T$  is defined with  $v = v'$ , except for

$$\begin{aligned} \sigma_{Var}(v') &= \epsilon_{var}(\langle stmt \rangle \hat{\ } prg, p_{vis}, s, \\ &\quad (subject_{var}(c), chan_{port}(c), subject_{port}(c)), \\ &\quad (modelTime(v), \sigma_{Var}(v), \kappa_{Chan}(v))) \\ \kappa_{Chan}(v') &= \epsilon_{chan}(\langle stmt \rangle \hat{\ } prg, p_{vis}, s, \\ &\quad (subject_{var}(c), chan_{port}(c), subject_{port}(c)), \\ &\quad (modelTime(v), \sigma_{Var}(v), \kappa_{Chan}(v))) \\ \kappa_{Subj_{prog}}(v') &= \kappa_{Subj_{prog}}(v) \oplus \{(s, op) \mapsto (\epsilon_{prg}(\langle stmt \rangle \hat{\ } prg, p_{vis}, s, \\ &\quad (subject_{var}(c), chan_{port}(c), subject_{port}(c)), \\ &\quad (modelTime(v), \sigma_{Var}(v), \kappa_{Chan}(v))), \delta, p_{vis})\} \text{ with } \delta \in \mathbb{R}_+ \quad \square \end{aligned}$$

The transition updates the local variable valuations and the channel valuations according to the effect of the statement. Further, the remaining program string, which is also part of the effect, is inserted for  $(s, op)$ .

For different kind of statements, the respective effects differ. In the following subsections, the definition of the effect function  $\epsilon$  is given incrementally, per statement.

#### Write Access to Channels

The *writing* of data into a channel is defined by the statement `put`.

**put.** The effect of a statement  $\text{put}(p_{write}, vis, data)$ , where local variables  $vis, x \in Var$  hold a visibility set  $visset \in VisibilitySet$  and a data value  $val \in Val$ , and  $p_{write} \in Port$  is a port which is accessible by subject  $s$ , i.e.  $s \in subject_{port}(c)(p_{write})$ , is defined by

$$\epsilon(\langle \text{put}(p_{write}, vis, x) \rangle \hat{\ } prg, param, s, c, v) = (prg, (\sigma_{Var}(v), \kappa_{Chan}(v) \oplus \{insert_{Port}(p_{write}, (data_{Val}(\sigma_{Var}(v)(x)), \sigma_{Var}(v)(vis)), c, v)\}))$$

On the one hand, the data is written to the channel through the given port, with the attached visibilities. On the other hand, the **put** statement is consumed and removed from the program string.

### Read Access to Channels

The *reading* of data from a channel is provided by different statements: (1) **get** retrieves the data value that is currently visible for a specific port. (2) **getTime** gets the publication time stamp for that data value. (3) **getPrevious** provides the second newest data value for a specific port. (4) **getPreviousTime** reads the publication time stamp for the second newest value.

**get.** A **get** statement is an assignment of the form  $x := \text{get}(p_{read})$  with a local variable  $x \in Var$  of subject  $s$ , i.e.  $subject_{var}(c)(x) = s$ , and a port  $p_{read} \in Port$  which is accessible by subject  $s$ , i.e.  $s \in subject_{port}(c)(p_{read})$ .

Then the current data value for the given port is retrieved and assigned to the local variable  $x$ , if  $x$  is of corresponding type, and the **get** statement is removed from the operation's program string:

$$\epsilon(\langle x := \text{get}(p_{read}) \rangle \hat{\ } prg, param, s, c, v) = (prg, (\sigma_{Var}(v) \oplus \{x \mapsto val_{Data}(data_{Port,cur}(p_{read}, c, v))\}, \kappa_{Chan}(v)))$$

The variable must have a fitting type, that is,  $val_{Data}(data_{Port,cur}(p_{read}, c, v)) \in type_{Var}(x)$ .

**getTime.** Similarly to the retrieval of the current data, the associated publication time stamp can be read. This may be used while calculating integration steps, when the evolution of a value wrt. time is needed. Additionally, the previous value and time also have effect on the calculation; these are given in the succeeding paragraphs.

The effect of a statement  $t := \text{getTime}(p_{read})$  is that the time stamp is read and assigned to a (correctly typed) local variable  $t \in Var$  of subject  $s$ , i.e.  $type_{Var}(t) \supseteq ModelTime$  and  $subject_{var}(c)(t) = s$ . The time stamp is read for a port  $p_{read} \in Port$  which is accessible by subject  $s$ , i.e.  $s \in subject_{port}(c)(p_{read})$ . Further, the **getTime** statement is removed from the operation's program string.

$$\epsilon(\langle t := \text{getTime}(p_{read}) \rangle \hat{\ } prg, param, s, c, v) = (prg, (\sigma_{Var}(v) \oplus \{t \mapsto tick_{Port,cur}(p_{read}, c, v)\}, \kappa_{Chan}(v)))$$

**getPrevious.** A **getPrevious** statement is an assignment which is very similar to the **get** assignment; the only difference is, that instead of the newest value for the given port, the *second* newest one is retrieved.

For a statement  $x := \text{getPrevious}(p_{read})$  with a local variable  $x \in Var$  of subject  $s$ , i.e.  $subject_{var}(c)(x) = s$ , and a port  $p_{read} \in Port$  accessible by subject  $s$ , i.e.  $s \in subject_{port}(c)(p_{read})$ , the previous data value of the current one for the given port is retrieved and assigned to the local variable  $x$ , and the `getPrevious` statement is removed from the operation's program string:

$$\begin{aligned} \epsilon(\langle x := \text{getPrevious}(p_{read}) \rangle \hat{\ } prg, param, s, c, v) = \\ (prg, (\sigma_{Var}(v) \oplus \{x \mapsto val_{Data}(data_{Port,prev}(p_{read}, c, v))\}, \kappa_{Chan}(v))) \end{aligned}$$

The variable must have a fitting type, that is,  $val_{Data}(data_{Port,prev}(p_{read}, c, v)) \in type_{Var}(x)$ .

**getPreviousTime.** The time stamp of the second newest value is accessed in the analogous way as the time stamp of the current value is.

The effect of a statement  $t := \text{getPreviousTime}(p_{read})$  is that the time stamp is read and assigned to the a local variable  $t \in Var$  of correct type and of subject  $s$ , i.e.  $type_{Var}(t) \supseteq ModelTime$  and  $subject_{var}(c)(t) = s$ . The time stamp is read for a port  $p_{read} \in Port$  which is accessible by subject  $s$ , i.e.  $s \in subject_{port}(c)(p_{read})$ . Further, the `getPreviousTime` statement is removed from the operation's program string.

$$\begin{aligned} \epsilon(\langle t := \text{getPreviousTime}(p_{read}) \rangle \hat{\ } prg, param, s, c, v) = \\ (prg, (\sigma_{Var}(v) \oplus \{t \mapsto tick_{Port,prev}(p_{read}, c, v)\}, \kappa_{Chan}(v))) \end{aligned}$$

### Read Access to Model Time

The statement `getCurrentTime` retrieves the model time for use in the program:

**getCurrentTime.** The effect of a statement  $t := \text{getCurrentTime}()$  is that the current model time is read and assigned to a local variable  $t \in Var$ , which is accessible for subject  $s$ , i.e.  $subject_{var}(c)(t) = s$ , supposed that  $type_{Var}(t) \supseteq ModelTime$ . Anyway, the `getCurrentTime` statement is removed from the operation's program string.

$$\begin{aligned} \epsilon(\langle t := \text{getCurrentTime}() \rangle \hat{\ } prg, param, s, c, v) = \\ (prg, (\sigma_{Var}(v) \oplus \{t \mapsto modelTime(v)\}, \kappa_{Chan}(v))) \end{aligned}$$

### Read Access to the Visibility Set Parameter

The statement `getVisParam` retrieves the program's visibility set parameter:

**getVisParam.** The effect of a statement of the form  $vis := \text{getVisParam}()$  is that the visibility set parameter is read and assigned to the local variable  $vis \in Var$  of subject  $s$ , i.e.  $subject_{var}(c)(vis) = s$ . The type of  $vis$  must fit, and a parameter value must be available:  $type_{Var}(vis) \supseteq VisibilitySet$  and  $param \neq \lambda$ . Further, the `getVisParam` statement is removed from the operation's program string:

$$\begin{aligned} \epsilon(\langle vis := \text{getVisParam}() \rangle \hat{\ } prg, param, s, c, v) = \\ (prg, (\sigma_{Var}(v) \oplus \{vis \mapsto param\}, \kappa_{Chan}(v))) \end{aligned}$$

### The Operational Rules of Standard Commands

In this section, we briefly list the effects of the conventional statements of while-programs, as they are discussed in detail in [AO97]. We do *not* repeat the semantics of arithmetic and boolean expressions, but assume that they are known, along with the notations used in [AO97].

**Assignment.** An assignment statement evaluates an expression  $exp$  and assigns the value to a variable  $x \in Var$  of subject  $s$ , i.e.  $subject_{var}(c)(x) = s$ . The statement is removed from the program string.

$$\epsilon(\langle x := exp \rangle \hat{\ } prg, param, s, c, v) = (prg, (\sigma_{Var}(v) \oplus \{x \mapsto \sigma(exp)\}, \kappa_{Chan}(v)))$$

**Conditional Statement.** A conditional statement chooses one of two (sub-)programs  $sub_1$  and  $sub_2$ , depending on the valuation of a boolean expression  $bexp$ . Then the effect of the statement is defined as

$$\epsilon(\langle \text{if } (bexp) \{sub_1\} \text{ else } \{sub_2\} \rangle \hat{\ } prg, param, s, c, v) = \begin{cases} (sub_1 \hat{\ } prg, (\sigma_{Var}(v), \kappa_{Chan}(v))) & \text{if } \sigma_{Var}(v) \models bexp \\ (sub_2 \hat{\ } prg, (\sigma_{Var}(v), \kappa_{Chan}(v))) & \text{if } \sigma_{Var}(v) \models \neg bexp \end{cases}$$

**While-Loop Statement.** A while-loop repeatedly executes a (sub-)program  $sub$ , as long as a boolean expression  $bexp$  evaluates to *true*. Then the effect of the statement is given by

$$\epsilon(\langle \text{while } (bexp) \{sub\} \rangle \hat{\ } prg, param, s, c, v) = \begin{cases} (sub \hat{\ } \langle \text{while } (bexp) \{sub\} \rangle \hat{\ } prg, (\sigma_{Var}(v), \kappa_{Chan}(v))) & \text{if } \sigma_{Var}(v) \models bexp \\ (prg, (\sigma_{Var}(v), \kappa_{Chan}(v))) & \text{if } \sigma_{Var}(v) \models \neg bexp \end{cases}$$

### Set Operations

We introduce special statements to support the handling of set values within programs. These are the adding and retrieving of entries, as well as the clearing of sets and the reading of the set size.

**clear.** For a variable  $s \in Var$  that can have the empty set as current value, i.e. with  $\emptyset \in type_{Var}(s)$ , the operation **clear** assigns it:

$$\epsilon(\langle \text{clear}(s) \rangle \hat{\ } prg, param, s, c, v) = (prg, (\sigma_{Var}(v) \oplus \{s \mapsto \emptyset\}, \kappa_{Chan}(v)))$$

**addEntry.** For a variable  $s \in Var$  with a current set value, i.e.  $\exists S \bullet \sigma_{Var}(v)(s) \in \mathcal{P}(S)$ , the operation **addEntry** adds the value given by the variable  $x \in Var$ :

$$\epsilon(\langle \text{addEntry}(s, x) \rangle \hat{\ } prg, param, s, c, v) = (prg, (\sigma_{Var}(v) \oplus \{s \mapsto \sigma_{Var}(v)(s) \cup \{\sigma_{Var}(v)(x)\}\}, \kappa_{Chan}(v)))$$

This is only defined, if the new set also fits to the variable's type, that means if  $\sigma_{Var}(v)(s) \cup \{\sigma_{Var}(v)(x)\} \in type_{Var}(s)$ .

**getEntry.** The reading of entries from a set value is defined by the statement **getEntry**, such that one element of the set of the variable  $s \in Var$  is assigned to variable  $x \in Var$ .

For this, a mapping  $\text{anyseq}_{SET} : \mathcal{P}(SET) \rightarrow \text{seq } SET$  is supposed to define an arbitrary sequence for all elements of a given finite set  $set \subseteq SET$ , i.e.  $\forall set \subseteq SET \bullet |set| \in \mathbb{N}_0 \Rightarrow \text{ran}(\text{anyseq}_{SET}(set)) = set \wedge |\text{anyseq}_{SET}(set)| = |set|$ .

Then, the variable  $i \in Var$  defines an index of this sequence, which identifies the element to be read:

$$\epsilon(\langle x := \text{getEntry}(s, i) \rangle \wedge \text{prg}, param, s, c, v) = (\text{prg}, (\sigma_{Var}(v) \oplus \{x \mapsto \text{anyseq}_{\sigma_{Var}(v)(s)}(\sigma_{Var}(v)(s))(\sigma_{Var}(v)(i) + 1)\}, \kappa_{Chan}(v)))$$

The variable  $i$  must hold an index that maps to a set entry:  $\sigma_{Var}(v)(i) \in \mathbb{N}_0 \wedge |\sigma_{Var}(v)(s)| > \sigma_{Var}(v)(i)$ . Finally, the resulting set must fit to the type of  $x$ :  $\text{anyseq}_{\sigma_{Var}(v)(s)}(\sigma_{Var}(v)(s))(\sigma_{Var}(v)(i) + 1) \in \text{type}_{Var}(x)$

**size.** The statement **size** accesses the size of a set value that is held by a variable  $s \in Var$ , and assigns it to a variable  $x \in Var$ , provided that  $\text{type}_{Var}(x) \supseteq \mathbb{N}_0$ :

$$\epsilon(\langle x := \text{size}(s) \rangle \wedge \text{prg}, param, s, c, v) = (\text{prg}, (\sigma_{Var}(v) \oplus \{x \mapsto |\sigma_{Var}(v)(s)|\}, \kappa_{Chan}(v)))$$

### Pair Operations

In order to read and modify pairs of values within a program, the operations **left**, **right**, **setLeft**, and **setRight** are defined:

**left.** Read access to the first projection of a pair of values is provided by the statement **left** for a variable  $p \in Var$  with  $\exists(l, r) \in (L \times R) \bullet \sigma_{Var}(v)(p) = (l, r)$ . The value is then stored in a variable  $x \in Var$  of corresponding type, i.e. with  $l \in \text{type}_{Var}(x)$ :

$$\epsilon(\langle x := \text{left}(p) \rangle \wedge \text{prg}, param, s, c, v) = (\text{prg}, (\sigma_{Var}(v) \oplus \{x \mapsto \pi_1 \sigma_{Var}(v)(p)\}, \kappa_{Chan}(v)))$$

**right.** Read access to the second projection of a pair of values is provided analogously to **left**, with the corresponding assumptions.

$$\epsilon(\langle x := \text{right}(p) \rangle \wedge \text{prg}, param, s, c, v) = (\text{prg}, (\sigma_{Var}(v) \oplus \{x \mapsto \pi_2 \sigma_{Var}(v)(p)\}, \kappa_{Chan}(v)))$$

**setLeft.** Write access to the first projection of a pair of values is provided by the statement **setLeft** for variables  $p, x \in Var$ :

$$\epsilon(\langle \text{setLeft}(p, x) \rangle \wedge \text{prg}, param, s, c, v) = (\text{prg}, (\sigma_{Var}(v) \oplus \{p \mapsto (\sigma_{Var}(v)(x), \pi_2 \sigma_{Var}(v)(p))\}, \kappa_{Chan}(v)))$$

The previous value of  $p$  must be a pair already:  $\exists(l, r) \in (L \times R) \bullet \sigma_{Var}(v)(p) = (l, r)$ . The new value must fit to the type of  $p$  again:  $(\sigma_{Var}(v)(x), \pi_2 \sigma_{Var}(v)(p)) \in \text{type}_{Var}(p)$ .

**setRight.** Write access to the second projection of a pair of values is provided by the statement **setRight** for variables  $p, x \in Var$ :

$$\epsilon(\langle \mathbf{setRight}(p, x) \rangle \hat{\ } prg, param, s, c, v) = \\ (\langle prg, (\sigma_{Var}(v) \oplus \{p \mapsto (\pi_1 \sigma_{Var}(v)(p), \sigma_{Var}(v)(x)\}), \kappa_{Chan}(v)) \rangle)$$

The previous value of  $p$  must be a pair already, and the new value must fit to the type of  $p$  again:  $\exists(l, r) \in (L \times R) \bullet \sigma_{Var}(v)(p) = (l, r)$  and  $(\pi_1 \sigma_{Var}(v)(p), \sigma_{Var}(v)(x)) \in type_{Var}(p)$ .

### Non-Determinism

The statement **random** provides non-determinism by assigning a random natural number (or 0) to a local variable.

**random.** A **random** statement is an assignment  $x := \mathbf{random}()$  of a local variable  $x \in Var$ , which is accessible for subject  $s$ , i.e.  $subject_{var}(c)(x) = s$ . The assigned value is an arbitrary natural number or 0. The **random** statement is consumed.

$$\exists val \in \mathbb{N}_0 \bullet \epsilon(\langle x := \mathbf{random}() \rangle \hat{\ } prg, param, s, c, v) = \\ (\langle prg, (\sigma_{Var}(v) \oplus \{x \mapsto val\}), \kappa_{Chan}(v)) \rangle)$$

The variable  $x$  must be of corresponding type:  $val \in type_{Var}(x)$

### 4.6.3 Program Termination

The termination of a complete program, and therefore of the associated operation, takes place when either the program string is empty, or the explicit **return** is reached. Since the termination itself may also take some (small amount of) time, it is treated in a similar way as statements are.

Suppose that in state  $(c, v) \in S$  an operation  $(s, op) \in \text{dom } \kappa_{Subj_{prog}}(v)$  with an empty program string or a program string beginning with **return** has an elapsed execution time:

$$\kappa_{Subj_{prog}}(v)(s, op) = (prg, 0, p_{vis}) \\ \text{with } prg = \langle \rangle \vee head(prg) = \mathbf{return}$$

As a result, the remaining program string is empty:

$$\epsilon(prg, param, s, c, v) = \\ (\langle \rangle, (\sigma_{Var}(v), \kappa_{Chan}(v)))$$

Further, the corresponding LWP is released, i.e. it is controlled by the scheduler now. Additionally, the execution time is unset, and the visibility set parameter is removed.

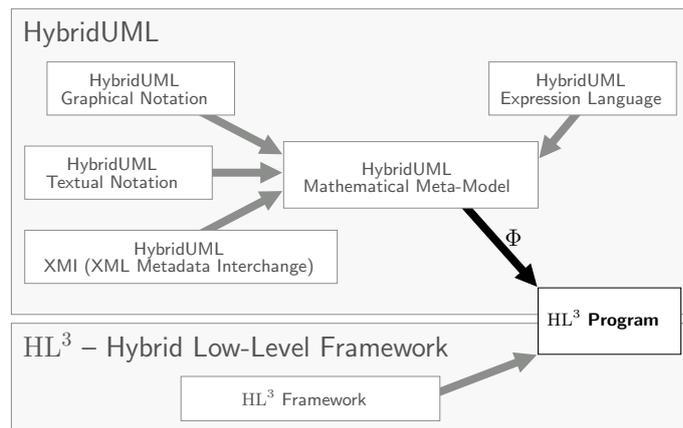
**Rule 4.6.4** Therefore, we have a transition  $(c, v) \longrightarrow (c, v') \in T$  with  $v = v'$ , except for

$$\kappa_{Subj_{prog}}(v') = \kappa_{Subj_{prog}}(v) \oplus \{(s, op) \mapsto (\epsilon_{prg}(prg, p_{vis}, s, \\ (subject_{var}(c), chan_{port}(c), subject_{port}(c)), \\ (modelTime(v), \sigma_{Var}(v), \kappa_{Chan}(v))), \infty, \lambda)\} \\ \kappa_{LWP}(v') = \kappa_{LWP}(v) \oplus \{p \mapsto scheduler \mid \kappa_{LWP}(v)(p) = s\} \quad \square$$



## Chapter 5

# Executable HybridUML Semantics: Transformation Definition



This chapter provides the specific transformation  $\Phi_{HUML}$  from HybridUML models to instances of the HL<sup>3</sup> design pattern. The transformation is defined formally, therefore the HybridUML executable semantics results. The resulting semantics is the HybridUML simulation semantics – it defines the behavior of a self-contained simulation of the complete HybridUML model.

The transformation results are separated into two parts: (1) Independently from the specific model, HybridUML-specific behavior definitions are given. (2) Corresponding to the specific HybridUML model, the entities of HL<sup>3</sup> model, as well as their dependencies, are defined.

Explicit programs are part of the resulting HL<sup>3</sup> model, they are defined by HybEL expressions contained in the HybridUML model. The transformation rules for expressions into programs are presented separately.

For the definition of the static structure of the HL<sup>3</sup> model, an evaluation semantics of HybEL expressions in the context of given HybridUML models is defined.

This chapter defines the transformation  $\Phi_{HUMML}$  of HybridUML specifications into HL<sup>3</sup> models. Since HL<sup>3</sup> models have a formal operational semantics (defined in chapter 4), particular HybridUML specifications *spec* obtain their semantics directly by  $\Phi_{HUMML}$ .

The result of  $\Phi_{HUMML}$  consists of two parts: (1) From the HybridUML specification *spec*, the constant part  $c_{spec} \in CONST$  of the state space  $S$  of the HL<sup>3</sup> model is defined. This includes the definition of abstract machines, transitions, and flows from the HybridUML agents and modes with their contained expressions and (HybridUML) transitions. (2) Independently from the particular specification *spec*, HybridUML-specific definitions for operations of abstract subjects (and their internal state) are given, i.e. the operations that define the behavior of *Abstract Machines* and the *Selector*.

By the transformation  $\Phi_{HUMML}$ , a *simulation of the complete HybridUML specification* is defined. We do not consider any architectural specification (see section 1.1). Therefore, (1) all HybridUML agents from the specification are mapped to abstract machines. There are *no interface modules*, because the simulation does not interact with an external environment. An architectural specification would be used to define parts of the specification that are defined externally, such that the respective agent instances would be transformed into interface modules, rather than into abstract machines.

Further, (2) the selector defined in this chapter is the *HybridUML Simulation Selector*. If in contrast, specific executions of the HybridUML specification had to be chosen, for example for a test setting (with external components) that utilizes elaborate test data generation algorithms, a different selector would be defined.

This chapter is structured as follows: Sections 5.1, 5.2, and 5.3 define some prerequisites for the definitions of the HL<sup>3</sup> model, as well as for the abstract subject's operations. These are (1) the notion of an *intermediate representation* of HybridUML specifications, (2) the definition of the *evaluation semantics* of HybEL expressions in the context of the intermediate HybridUML representation, (3) and the definition of several mappings on expressions and expression nodes for code generation.

In section 5.4, the constant state space  $c_{spec}$  of the HL<sup>3</sup> model is defined according to HybridUML specifications *spec*, which are given syntactically as discussed in chapter 2. The creation of programs  $p \in Program$ , which define a significant part of the HL<sup>3</sup> model's behavior, is presented separately in section 5.5. Examples of the resulting HL<sup>3</sup> model are given by means of references to appendix D, which contains the C++ variant of the HL<sup>3</sup> program of the Radio-Based Train Control case study, as it is generated by the implementation of the transformation  $\Phi_{HUMML}$ .

Finally, the HybridUML-specific definition of the operations and the internal state of abstract subjects is discussed in section 5.6.

## 5.1 Intermediate Specification Representation

In this section, the HybridUML specification is transformed into an intermediate representation that consists of:

*Tree of Agent Instance Nodes* From the agent and agent instance specifications, a tree of agent instance nodes is created. Agent instance nodes are the

resulting objects that are derived from recursive application of agent and agent instance specifications.

*Set of Basic Agent Instance Nodes* The leafs of the tree of agent instance nodes are the active objects that encapsulate the sequential behavioral components of the system.

*Sets of Property and Signal Nodes* For each agent instance node, a set of property nodes and a set of signal nodes is derived from its agent's properties. These represent instances of properties and signals wrt. the agent instance node.

*Sets of Connected Property Nodes and Connected Signal Nodes* The maximal sets of connected property nodes and connected signal nodes, as defined by connectors between properties and signals from the HybridUML specification, represent the shared variables of the system.

*Tree of Mode Instance Nodes* Each basic agent instance node has an attached behavioral specification which is defined by a tree of mode instance nodes. From the top-level mode instance of the associated agent, this tree is derived.

*Sets of Control Point Instance Nodes and Sets of Transition Nodes* The mode instance nodes are connected by transition nodes via control point instance nodes. The control point instance nodes are instantiated from control point instances and control points of the respective mode instances and modes, the transition nodes represent the transitions from the HybridUML specification which connect them.

*Sets of Expression Nodes* Attached to mode instance nodes and transition nodes, there are expression nodes that represent the expressions of the original specification: triggers, guards, and actions for transitions, and flow constraints as well as invariant constraints for modes.

The remainder of this section is divided into structural aspects (tree of agent instance nodes, property and signal nodes, and the connected sets thereof) and behavioral aspects (tree of mode instance nodes with control point instance nodes, transition nodes, and expression nodes).

### 5.1.1 Structure

**Agent Instance Nodes.** In contrast to agent instances  $ai \in AI$  which define sets of instances per containing agent, an agent instance node represents exactly one single instance of an agent within the system. The possible agent instance nodes are defined as:

$$AIN = \text{seq } AIN \times AI \times \mathbb{N}_0$$

Each of these represents an agent instance, given by its second component

$$ai_{AIN} = \pi_2 AIN$$

In order to identify the agent instance node  $ain$  uniquely, the context of the represented agent instance is contained as a path  $\langle a_1, \dots, a_n \rangle \in \text{seq } AIN$  of agent

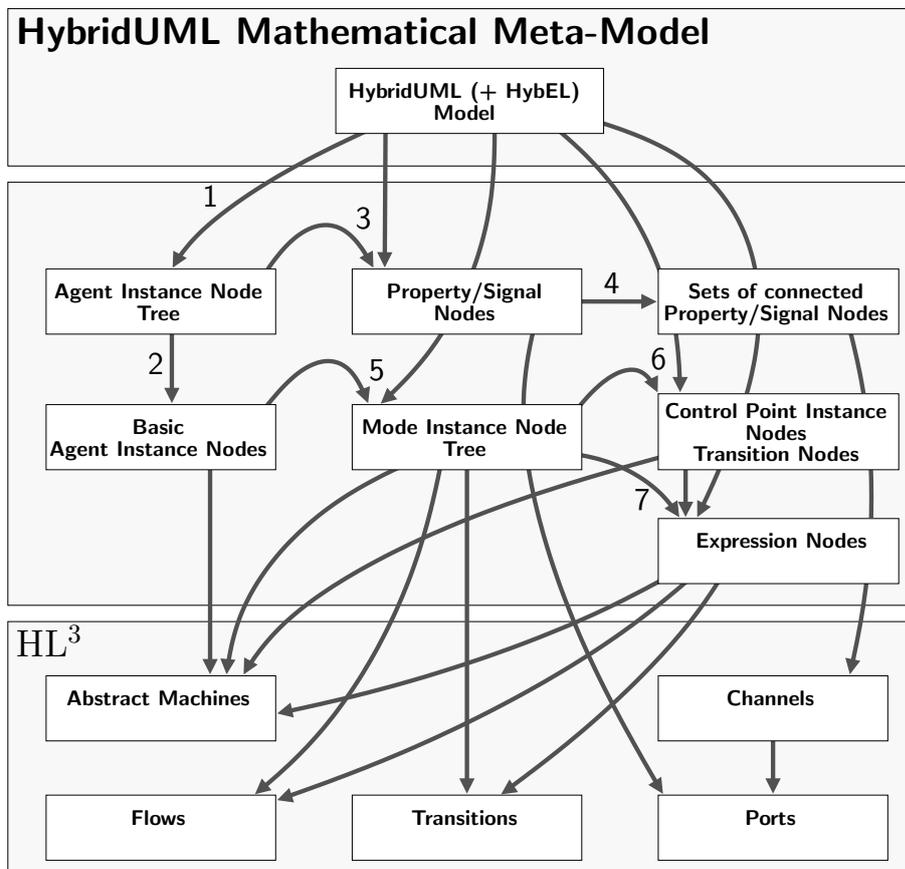


Figure 5.1: Simplified transformation illustration. From the HybridUML specification, an HL<sup>3</sup> model is created, via the Intermediate Representation.

instance nodes leading from its direct ancestor  $a_1$  to a top-level agent instance node  $a_n$ . The top-level node itself contains the empty context  $\langle \rangle \in \text{seq } AIN$ . The context is given as the first component

$$path_{AIN} = \pi_1 AIN$$

Within the context of its parent agent, the represented agent instance has a multiplicity  $m$ . This shall define the number of agent instance nodes that represent the agent instance. Each of these agent instance nodes is supposed to represent one index of  $[0, m - 1]$ :

$$index_{AIN} = \pi_3 AIN$$

For a particular HybridUML specification, there is a subset  $AIN_{spec} \subseteq AIN$  of agent instance nodes that correspond to the specification. These agent instance nodes contain property nodes that shall represent a single index of a property of the corresponding agent, that means, properties of the agent with multiplicity  $m$  shall be represented by  $m$  independent property nodes:

$$vn_{AIN_{spec}} : AIN_{spec} \rightarrow \mathcal{P}(VN_{spec})$$

The same holds for signal nodes:

$$sn_{AIN_{spec}} : AIN_{spec} \rightarrow \mathcal{P}(SN_{spec})$$

The definitions are given below, with the definitions of property and signal nodes.

**Tree of Agent Instance Nodes.** The mapping  $tree_{AIN}$  defines how an agent instance node is (recursively) mapped to a tree<sup>1</sup> of agent instances nodes  $Tree_{AIN}$ , such that the resulting tree represents the complete structural specification that is contained in the corresponding agent instance:

$$\begin{aligned} Tree_{AIN} &= \text{tree } AIN \\ tree_{AIN} &: AIN \rightarrow Tree_{AIN} \\ (p, ai, n) &\mapsto \\ &((p, ai, n), \{tree_{AIN}((p_s, ai_s, n_s)) \mid ai_s \in part_A(agent_{AI}(ai)) \\ &\wedge 0 \leq n_s < eval(mult_{AI}(ai_s), head(p)) \wedge p_s = \langle(p, ai, n)\rangle \frown p\}) \\ &; p \neq \langle\rangle \\ \langle\rangle, ai, n &\mapsto \\ &((\langle\rangle, ai, n), \{tree_{AIN}((p_s, ai_s, n_s)) \mid ai_s \in part_A(agent_{AI}(ai)) \\ &\wedge 0 \leq n_s < eval_{\emptyset}(mult_{AI}(ai_s)) \wedge p_s = \langle(\langle\rangle, ai, n)\rangle \frown \langle\rangle\}) \end{aligned}$$

The definition of the evaluation function  $eval$  of HybEL expressions in the context of agent instance nodes is postponed to section 5.2.

In order to obtain the tree of agent instance nodes for the complete specification, the mapping  $tree_{AIN}$  is used with the single top-level agent instance that represents the system:

$$\begin{aligned} tree_{AI} &: AI \rightarrow Tree_{AIN} \\ ai &\mapsto tree_{AIN}(\langle\rangle, ai, 0) \end{aligned}$$

Thus,  $tree_{AI}(ai_{tl})$  is the complete agent instance node tree for the specification. The set of all contained trees is:

$$Tree_{AIN_{spec}} = \{\text{subtrees}_{AIN}(tree_{AI}(ai_{tl}))\}$$

The set of agent instance nodes that are constructed for the specification then is:

$$AIN_{spec} = \{ain \in AIN \mid \exists(ain_1, \{t_0, \dots, t_k\}) \in Tree_{AIN_{spec}} \bullet ain_1 = ain\}$$

The leafs of the tree of agent instance nodes are the basic agent instance nodes, they define the sequential behavioral components of the resulting system:

$$AIN_{basic} = \{ain \in AIN_{spec} \mid tree_{AIN}(ain) = (ain, \emptyset)\}$$

<sup>1</sup>See chapter A for the definition of trees.

**Property Nodes.** A property node represents an index of a property within the context of an agent instance node. The set of all possible property nodes is

$$VN = AIN \times V \times (P_V \cup \{\lambda\}) \times (PI_V \cup \{\lambda\}) \times \mathbb{N}_0$$

The first component is embedding agent instance node:

$$ain_{VN} = \pi_1 VN$$

The represented variable is contained, too:

$$var_{VN} = \pi_2 VN$$

If a corresponding variable port instance exists, it shall be associated with the node, otherwise the special value  $\lambda$  denotes the absence of a port instance:

$$portIns_{VN} = \pi_4 VN$$

If a corresponding variable port exists, it shall be associated with the node, similarly to port instances:

$$port_{VN} = \pi_3 VN$$

Finally, the node's index wrt. the property's multiplicity is contained:

$$index_{VN} = \pi_5 VN$$

The subset  $VN_{spec} \subseteq VN$  defines the property nodes which are part of the representation of a concrete specification.

**Consecutive Index of Property Nodes.** In order to realize *point-to-point* connections between port instances of multiplicity  $m_p$ , contained by agent instances with multiplicity  $m_a$ , every property node is supposed to have a *consecutive index* within the context of the corresponding agent.

For the calculation of the consecutive index, a mapping is provided that gives all property nodes for a single one, such that all property nodes coincide for the port instance they represent:

$$\begin{aligned} vn_{VN,PI_V} : VN &\rightarrow \mathcal{P}(VN) \\ (ain, v, p, pi, n) &\mapsto \{(ain, v, p, pi, m) \in VN\}; pi \neq \lambda \\ (ain, v, p, \lambda, n) &\mapsto \emptyset \end{aligned}$$

Further, a sequence of property nodes, ordered by the properties' indices they represent, is defined:

$$\begin{aligned} vnseq : \mathcal{P}(VN) &\rightarrow \text{seq } VN \\ &\text{such that} \\ \forall vn \in \mathcal{P}(VN) \bullet |vnseq(vn)| &= |vn| \wedge \text{ran } vnseq(vn) = vn \\ &\text{and} \\ \forall i_1 \mapsto vn_1, i_2 \mapsto vn_2 \in &vnseq(vn) \bullet i_1 \leq i_2 \Rightarrow index_{VN}(vn_1) \leq index_{VN}(vn_2) \end{aligned}$$

The *local consecutive index* of a property node is then the (unique) position of the property node within the sequence of property nodes that represent its port index:

$$\begin{aligned} cindex_{VN,loc} : VN &\rightarrow \mathbb{N}_0 \\ vn &\mapsto i; i \mapsto vn \in vnseq(vn_{VN,PI_V}(vn)) \end{aligned}$$

Finally, the consecutive index then is calculated from the agent instance's index and the property node's local consecutive index:

$$\begin{aligned} cindex_{VN} : VN &\rightarrow \mathbb{N}_0 \\ vn &\mapsto index_{AIN}(ain_{VN}(vn)) \cdot |vn_{VN,PI_V}(vn)| + cindex_{VN,loc}(vn) \end{aligned}$$

**Signal Nodes.** A signal node represents an index of a signal within the context of an agent instance node, exactly in the same fashion as variable nodes do for properties. Therefore, the set of all possible signal nodes is

$$SN = AIN \times S \times (P_S \cup \{\lambda\}) \times (PI_S \cup \{\lambda\}) \times \mathbb{N}_0$$

The components agent instance node, signal, variable port instance, variable port, and index are accessible by the following projections:

$$\begin{aligned} ain_{SN} &= \pi_1 SN \\ sig_{SN} &= \pi_2 SN \\ portIns_{SN} &= \pi_4 SN \\ port_{SN} &= \pi_3 SN \\ index_{SN} &= \pi_5 SN \end{aligned}$$

The subset  $SN_{spec} \subseteq SN$  then defines the signal nodes that are part of the representation of a concrete specification.

**Consecutive Index of Signal Nodes.** Point-to-point connections are supported, too, by the definition of the consecutive index for signal nodes. This is done in the same fashion as for property nodes.

A mapping is provided that gives all signal nodes that coincide wrt. the port instance with a given one:

$$\begin{aligned} sn_{SN,PI_S} : SN &\rightarrow \mathcal{P}(SN) \\ (ain, s, p, pi, n) &\mapsto \{(ain, s, p, pi, m) \in SN\}; pi \neq \lambda \\ (ain, s, p, \lambda, n) &\mapsto \emptyset \end{aligned}$$

A sequence of signal nodes is defined:

$$\begin{aligned} snseq : \mathcal{P}(SN) &\rightarrow \text{seq } SN \\ &\text{such that} \\ \forall sn \in \mathcal{P}(SN) \bullet |snseq(sn)| &= |sn| \wedge \text{ran } snseq(sn) = sn \\ &\text{and} \\ \forall i_1 \mapsto sn_1, i_2 \mapsto sn_2 \in snseq(sn) \bullet i_1 \leq i_2 &\Rightarrow index_{SN}(sn_1) \leq index_{SN}(sn_2) \end{aligned}$$

The *local consecutive index* of a signal node is:

$$\begin{aligned} cindex_{SN,loc} : SN &\rightarrow \mathbb{N}_0 \\ sn &\mapsto i; i \mapsto sn \in snseq(sn_{SN,PI_S}(sn)) \end{aligned}$$

The consecutive index then is calculated from the agent instance's index and the signal node's local consecutive index:

$$\begin{aligned} cindex_{SN} : SN &\rightarrow \mathbb{N}_0 \\ sn &\mapsto index_{AIN}(ain_{SN}(sn)) \cdot |sn_{SN,PI_S}(sn)| + cindex_{SN,loc}(sn) \end{aligned}$$

**Mapping to Property Nodes.** Based on the tree of agent instance nodes, the mapping  $vn_{AIN}$  collects the property nodes that represent the properties of the agents within each context of their agent instance nodes:

$$\begin{aligned}
vn_{AIN} &: Tree_{AIN} \rightarrow \mathcal{P}(VN) \\
((p, ai, n), \{t_0, \dots, t_k\}) &\mapsto \\
&\{((p, ai, n), v, pt, pti, m) \mid pti \in port_{Ins_{AI, Var}}(ai) \\
&\wedge m \in eval(indices_{PI_V}(pti), (p, ai, n)) \\
&\wedge pt = port_{PI_V}(pti) \\
&\wedge v = var_{P_V}(pt) \wedge 0 \leq m < eval(mult_V(v), (p, ai, n))\} \\
\cup &\{((p, ai, n), v, pt, \lambda, m) \mid \exists pti \in port_{Ins_{AI, Var}}(ai) \bullet \\
&(m \in eval(indices_{PI_V}(pti), (p, ai, n)) \\
&\wedge pt = port_{PI_V}(pti)) \\
&\wedge pt \in port_{A, Var}(agent_{AI}(ai)) \\
&\wedge v = var_{P_V}(pt) \wedge 0 \leq m < eval(mult_V(v), (p, ai, n))\} \\
\cup &\{((p, ai, n), v, \lambda, \lambda, m) \mid \exists pt \in port_{A, Var}(agent_{AI}(ai)) \bullet v = var_{P_V}(pt) \\
&\wedge v \in var_A(agent_{AI}(ai)) \\
&\wedge 0 \leq m < eval(mult_V(v), (p, ai, n))\} \\
\cup &\bigcup_{i=0}^k vn_{AIN}(t_i)
\end{aligned}$$

The property nodes which are part of the representation of the specification are the ones provided by  $vn_{AIN}$ , restricted to the agent instance nodes from the specification:

$$VN_{spec} = \{vn \in VN \mid \exists t \in Tree_{AIN_{spec}} \bullet vn \in vn_{AIN}(t)\}$$

The mapping  $vn_{AIN_{spec}}$  from agent instance nodes to sets of property nodes is directly given by  $vn_{AIN}$  as well:

$$\begin{aligned}
vn_{AIN_{spec}} &: AIN_{spec} \rightarrow \mathcal{P}(VN_{spec}) \\
ain &\mapsto vn_{AIN}(tree_{AIN}(ain))
\end{aligned}$$

For property nodes that represent parameters, the dedicated value defined by  $\sigma_V$  is evaluated, such that the value can be used while constructing the agent instance node tree. The special “ID” parameters are set to the agent instance node’s index:

$$\begin{aligned}
\sigma_{VN_{spec}} &: VN_{spec} \rightarrow VAL_{eval} \cup \{\lambda\} \\
(ain, v, pt, pti, m) &\mapsto eval(exp_{\sigma_V}(val), ain) \\
&; val \in \sigma_V \wedge var_{\sigma_V}(val) = v \\
&\wedge \exists a \in A \bullet v \in param_A(a) \wedge v \neq v_{id, agent_{AI}(ai)} \\
((p, ai, n), v, pt, pti, m) &\mapsto n; v = v_{id, agent_{AI}(ai)} \\
(ain, v, pt, pti, m) &\mapsto \lambda; \text{ else}
\end{aligned}$$

**Mapping to Signal Nodes.** Based on the tree of agent instance nodes, the mapping  $sn_{AIN}$  collects the signal nodes that represent the properties of the agents within each context of their agent instance nodes:

$$\begin{aligned}
sn_{AIN} : Tree_{AIN} &\rightarrow \mathcal{P}(SN) \\
((p, ai, n), \{t_0, \dots, t_k\}) &\mapsto \\
&\{((p, ai, n), s, pt, pti, m) \mid pti \in portIns_{AI, Sig}(ai) \\
&\wedge m \in eval(indices_{PI_S}(pti), (p, ai, n)) \\
&\wedge pt = port_{PI_S}(pti) \\
&\wedge s = sig_{P_S}(pt) \wedge 0 \leq m < eval(mult_S(s), (p, ai, n))\} \\
\cup &\{((p, ai, n), s, pt, \lambda, m) \mid \nexists pti \in portIns_{AI, Sig}(ai) \bullet \\
&(m \in eval(indices_{PI_S}(pti), (p, ai, n)) \\
&\wedge pt = port_{PI_S}(pti)) \\
&\wedge pt \in port_{A, Sig}(agent_{AI}(ai)) \\
&\wedge s = sig_{P_S}(pt) \wedge 0 \leq m < eval(mult_S(s), (p, ai, n))\} \\
\cup &\{((p, ai, n), s, \lambda, \lambda, m) \mid \nexists pt \in port_{A, Sig}(agent_{AI}(ai)) \bullet s = sig_{P_S}(pt) \\
&\wedge s \in sig_A(agent_{AI}(ai)) \\
&\wedge 0 \leq m < eval(mult_S(s), (p, ai, n))\} \\
\cup &\bigcup_{i=0}^k sn_{AIN}(t_i)
\end{aligned}$$

The signal nodes which are part of the representation of the specification are the ones provided by  $sn_{AIN}$ , restricted to the agent instance nodes from the specification:

$$SN_{spec} = \{sn \in SN \mid \exists t \in Tree_{AIN_{spec}} \bullet sn \in sn_{AIN}(t)\}$$

The mapping  $sn_{AIN_{spec}}$  from agent instance nodes to sets of signal nodes is directly given by  $sn_{AIN}$  as well:

$$\begin{aligned}
sn_{AIN_{spec}} : AIN_{spec} &\rightarrow \mathcal{P}(SN_{spec}) \\
ain &\mapsto sn_{AIN}(tree_{AIN}(ain))
\end{aligned}$$

**Connected Property Nodes.** Sets of connected property nodes define shared variables of the system. In the first step, directly connected property nodes are calculated – each set of nodes is connected by a single connector:

$$\begin{aligned}
VN_{conn, local} = & \\
&\{VN_{sub} \in \mathcal{P}(VN_{spec}) \mid \exists c \in C_V \bullet \\
&\forall (ain_1, v_1, p_1, pi_1, n_1), (ain_2, v_2, p_2, pi_2, n_2) \in VN_{sub} \bullet \\
&(pi_1 \in portIns_{C_V}(c) \wedge pi_2 \in portIns_{C_V}(c) \\
&\wedge (kind_{C_V}(c) = ptp \Rightarrow \\
&\quad cindex_{VN}((ain_1, v_1, p_1, pi_1, n_1)) = \\
&\quad cindex_{VN}((ain_2, v_2, p_2, pi_2, n_2))))\} \\
\cup &\{\{vn\} \mid vn \in VN_{spec}\}
\end{aligned}$$

$VN_{conn,local}$  contains sets of property nodes from the specification that are connected locally, i.e. they represent properties of agent instances which are connected through port instances within a common parent agent. Note that this is guaranteed because connectors are restricted to connect port instances only locally.

The sets of property nodes are not maximal, particularly there is a singleton set for each property node from the specification. The different connector kinds are respected, in that only nodes with coinciding consecutive index are connected for point-to-point connectors, in contrast to multicast connectors.

$$\begin{aligned}
VN_{conn,cont} = & \\
& \{VN_{sub} \in \mathcal{P}(VN_{spec}) \mid \exists c \in C_V \bullet \\
& \forall (ain_1, v_1, p_1, pi_1, n_1), (ain_2, v_2, p_2, pi_2, n_2) \in VN_{sub} \bullet \\
& (p_1 \in port_{C_V}(c) \wedge pi_2 \in portIns_{C_V}(c) \\
& \wedge (kind_{C_V}(c) = ptp \Rightarrow \\
& \quad cindex_{VN}((ain_1, v_1, p_1, pi_1, n_1)) = \\
& \quad cindex_{VN}((ain_2, v_2, p_2, pi_2, n_2))))\}
\end{aligned}$$

$VN_{conn,cont}$  contains sets of property nodes that are connected across hierarchy levels. A connector can be attached to a port of its containing agent and thus connect the corresponding property node with property nodes of contained agent instances. Each set contains exactly two property nodes, one per hierarchy level.

The second step consists of joining the sets of property nodes such that every set of such sets is united which share a common property node. This constitutes *transitive connections* via several connectors.

$$\begin{aligned}
VN_{conn} = & \\
& \{VN_{sub} \in \mathcal{P}(VN_{spec}) \mid \exists n \in \mathbb{N}_0 \bullet \\
& (VN_{sub} = \bigcup_{i=0}^n VN_i \\
& \wedge \forall i \in \{0..n\} \bullet VN_i \in (VN_{conn,local} \cup VN_{conn,cont}) \\
& \wedge \bigcap_{i=0}^n VN_i \neq \emptyset)\}
\end{aligned}$$

Finally, from the sets of connected property nodes, all maximal sets are taken:

$$VN_{conn,max} = \{VN_{sub} \in VN_{conn} \mid \nexists VN_2 \in VN_{conn} \bullet VN_{sub} \subset VN_2\}$$

**Connected Signal Nodes.** Sets of connected signal nodes define shared variables of the system, in the same way that connected property nodes do. First, directly connected signal nodes are calculated:

$$\begin{aligned}
SN_{conn,local} = & \\
& \{SN_{sub} \in \mathcal{P}(SN_{spec}) \mid \exists c \in C_S \bullet \\
& \forall (ain_1, s_1, p_1, pi_1, n_1), (ain_2, s_2, p_2, pi_2, n_2) \in SN_{sub} \bullet
\end{aligned}$$

$$\begin{aligned}
& (pi_1 \in portIns_{C_S}(c) \wedge pi_2 \in portIns_{C_S}(c) \\
& \wedge (kind_{C_S}(c) = ptp \Rightarrow \\
& \quad cindex_{SN}((ain_1, s_1, p_1, pi_1, n_1)) = \\
& \quad \quad cindex_{SN}((ain_2, s_2, p_2, pi_2, n_2)))) \\
& \cup \{ \{sn\} \mid sn \in SN_{spec} \}
\end{aligned}$$

$SN_{conn,local}$  contains sets of signal nodes from the specification that are connected locally.

$$\begin{aligned}
SN_{conn,cont} = & \\
& \{ SN_{sub} \in \mathcal{P}(SN_{spec}) \mid \exists c \in C_S \bullet \\
& \forall (ain_1, s_1, p_1, pi_1, n_1), (ain_2, s_2, p_2, pi_2, n_2) \in SN_{sub} \bullet \\
& (p_1 \in port_{C_S}(c) \wedge pi_2 \in portIns_{C_S}(c) \\
& \wedge (kind_{C_S}(c) = ptp \Rightarrow \\
& \quad cindex_{SN}((ain_1, s_1, p_1, pi_1, n_1)) = \\
& \quad \quad cindex_{SN}((ain_2, s_2, p_2, pi_2, n_2)))) \}
\end{aligned}$$

$VN_{conn,cont}$  contains sets of signal nodes that are connected across hierarchy levels; each set contains exactly two signal nodes, one per hierarchy level.

The second step consists of joining the sets of signal nodes such that every set of such sets is united which share a common signal node.

$$\begin{aligned}
SN_{conn} = & \\
& \{ SN_{sub} \in \mathcal{P}(SN_{spec}) \mid \exists n \in \mathbb{N}_0 \bullet \\
& (SN_{sub} = \bigcup_{i=0}^n SN_i \\
& \wedge \forall i \in \{0..n\} \bullet SN_i \in (SN_{conn,local} \cup SN_{conn,cont}) \\
& \wedge \bigcap_{i=0}^n SN_i \neq \emptyset) \}
\end{aligned}$$

Finally, from the sets of connected signal nodes, all maximal sets are taken:

$$SN_{conn,max} = \{ SN_{sub} \in SN_{conn} \mid \nexists SN_2 \in SN_{conn} \bullet SN_{sub} \subset SN_2 \}$$

### 5.1.2 Behavior

**Mode Instance Nodes.** Each mode instance from the specification is represented by a mode instance node within a basic agent instance node. In order to identify the node, the path of ancestor mode instance nodes is contained:

$$MIN = seq\ MIN \times MI \times AIN_{basic}$$

The components are provided by the projections

$$\begin{aligned}
mi_{MIN} &= \pi_2\ MIN \\
path_{MIN} &= \pi_1\ MIN \\
ain_{MIN} &= \pi_3\ MIN
\end{aligned}$$

Similar to agent instance nodes, the hierarchy of mode instance nodes is represented by a tree structure:

$$\begin{aligned}
Tree_{MIN} &= \text{tree } MIN \\
tree_{MIN} &: MIN \rightarrow Tree_{MIN} \\
(p, mi, ain) &\mapsto \\
&((p, mi, ain), \{tree_{MIN}((p_s, mi_s, ain)) \mid mi_s \in \text{submode}_M(\text{mode}_{MI}(mi))\}) \\
&\wedge p_s = \langle (p, mi, ain) \rangle \smallfrown p \}
\end{aligned}$$

The behavior of a basic agent instance node is defined by a dedicated mode instance node that represents the mode instance of the corresponding agent:

$$\begin{aligned}
mtree_{AIN} &: AIN_{basic} \rightarrow Tree_{MIN} \\
ain &\mapsto tree_{MIN}(\langle \rangle, mi, ain) \\
&; mi \in \text{behavior}_A(\text{agent}_{AI}(ai_{AIN}(ain)))
\end{aligned}$$

This mapping is well-defined, since (1) mode instance  $mi$  always exists and (2) there is at most one. The corresponding agent for basic agent instance node  $ain$  has no contained agent instances, therefore (1) holds and there can be at most one mode instance for any agent, thus (2) follows.

The set of mode instance nodes that represent mode instances for the HybridUML specification and the corresponding trees are those that are attached to the basic agent instance nodes:

$$\begin{aligned}
Tree_{MIN,spec} &= \\
&\{t \in Tree_{MIN} \mid \exists ain \in AIN_{basic} \bullet t \in \text{subtrees}_{MIN}(mtree_{AIN}(ain))\} \\
MIN_{spec} &= \\
&\{min \in MIN \mid \exists (min_1, \{t_0, \dots, t_k\}) \in Tree_{MIN,spec} \bullet min_1 = min\}
\end{aligned}$$

**Control Point Instance Nodes.** There are control point instance nodes that represent the control point instances of the mode instances:

$$CPIN = MIN \times CPI$$

with projections

$$\begin{aligned}
min_{CPIN} &= \pi_1 CPIN \\
cpi_{CPIN} &= \pi_2 CPIN
\end{aligned}$$

The control point instance nodes are attached to the mode instance nodes:

$$\begin{aligned}
cpin_{MIN} &: Tree_{MIN} \rightarrow \mathcal{P}(CPIN) \\
((p, mi, am), \{t_0, \dots, t_k\}) &\mapsto \\
&\{(p, mi, am), cpi \mid cpi \in cpi_{MI}(mi)\} \cup \bigcup_{i=0}^k cpin_{MIN}(t_i)
\end{aligned}$$

The control point instance nodes from the specification are those that are attached to mode instance nodes from the specification:

$$CPIN_{spec} = \{cpin \in CPIN \mid \exists t \in Tree_{MIN,spec} \bullet cpin \in cpin_{MIN}(t)\}$$

**Transition Nodes.** Transition nodes are the representations of transitions in the context of mode instance nodes:

$$TN = MIN \times T$$

The components are given by

$$\begin{aligned} min_{TN} &= \pi_1 TN \\ trans_{TN} &= \pi_2 TN \end{aligned}$$

The transition nodes are attached to the mode instance nodes:

$$\begin{aligned} tn_{MIN} : Tree_{MIN} &\rightarrow \mathcal{P}(TN) \\ ((p, mi, am), \{t_0, \dots, t_k\}) &\mapsto \\ \{((pi, mi, am), trans) \mid trans &\in trans_M(mode_{MI}(mi))\} \\ \cup \bigcup_{i=0}^k tn_{MIN}(t_i) & \end{aligned}$$

The transition nodes from the specification are those that are contained in the mode instance nodes from the specification:

$$TN_{spec} = \{tn \in TN \mid \exists t \in Tree_{MIN,spec} \bullet tn \in tn_{MIN}(t)\}$$

Each transition node is connected with the control point instance node that represents its source:

$$\begin{aligned} src_{TN_{spec}} : TN_{spec} &\rightarrow CPIN_{spec} \\ (min, trans) &\mapsto (min, cpi); src_T(trans) = cp_{CPI}(cpi) \\ (min, trans) &\mapsto (min_s, cpi) \\ ; src_T(trans) &= cpi \wedge \\ \exists t, t_0, \dots, t_k &\in Tree_{MIN,spec}, st \in \mathcal{P}(Tree_{MIN,spec}), i \in 0..k \bullet \\ t = (min, \{t_0, \dots, t_k\}) &\wedge t_i = (min_s, st) \end{aligned}$$

As in the definition of transitions themselves, it is distinguished between transitions that originate from a control point or from a control point instance. In the first case, the control point belongs to the transition's parent mode itself (represented by  $min$ ), whereas in the latter case the control point instance is attached to one of the submodes (represented by  $min_s$ ).

The transition nodes are connected with the control point instance nodes that represent their target:

$$\begin{aligned} tar_{TN_{spec}} : TN_{spec} &\rightarrow CPIN_{spec} \\ (min, trans) &\mapsto (min, cpi); tar_T(trans) = cp_{CPI}(cpi) \\ (min, trans) &\mapsto (min_s, cpi) \\ ; tar_T(trans) &= cpi \wedge \\ \exists t, t_0, \dots, t_k &\in Tree_{MIN,spec}, st \in \mathcal{P}(Tree_{MIN,spec}), i \in 0..k \bullet \\ t = (min, \{t_0, \dots, t_k\}) &\wedge t_i = (min_s, st) \end{aligned}$$

**Expression Nodes.** Expressions are attached to transitions, to modes, or to agents. They represent one of (1) trigger, (2) guard, (3) action, (4) flow, (5) invariant constraint, and (6) init state constraint:

$$\begin{aligned}
ExpN &= TrgExpN \cup GrdExpN \cup ActExpN \cup FlowExpN \cup InvExpN \\
&\quad \cup IscExpN \\
TrgExpN &= TN \times Exp \times \{trg\} \\
GrdExpN &= TN \times Exp \times \{grd\} \\
ActExpN &= TN \times Exp \times \{act\} \\
FlowExpN &= MIN \times Exp \times \{flow\} \\
InvExpN &= MIN \times Exp \times \{inv\} \\
IscExpN &= AIN \times Exp \times \{isc\}
\end{aligned}$$

They are partitioned into transition expression nodes, mode expression nodes, and agent expression nodes:

$$\begin{aligned}
TExpN &= TrgExpN \cup GrdExpN \times ActExpN \\
MExpN &= FlowExpN \cup InvExpN \\
AExpN &= IscExpN
\end{aligned}$$

Transition expression nodes consist of a transition node, an expression, and an expression kind (or role):

$$\begin{aligned}
tn_{TExpN} &= \pi_1 TExpN \\
exp_{TExpN} &= \pi_2 TExpN \\
kind_{TExpN} &= \pi_3 TExpN
\end{aligned}$$

Mode expression nodes consist of a mode instance node, an expression, and an expression kind (or role):

$$\begin{aligned}
min_{MExpN} &= \pi_1 MExpN \\
exp_{MExpN} &= \pi_2 MExpN \\
kind_{MExpN} &= \pi_3 MExpN
\end{aligned}$$

Agent expression nodes consist of an agent instance node, an expression, and an expression kind (or role):

$$\begin{aligned}
ain_{AExpN} &= \pi_1 AExpN \\
exp_{AExpN} &= \pi_2 AExpN \\
kind_{AExpN} &= \pi_3 AExpN
\end{aligned}$$

Every expression node is – directly or indirectly – contained in an agent instance node. For convenience, a respective mapping is defined:

$$\begin{aligned}
ain_{ExpN} : ExpN &\rightarrow AIN \\
expn &\mapsto \begin{cases} ain_{MIN}(min_{MExpN}(expn)); & expn \in MExpN \\ ain_{MIN}(min_{TN}(tn_{TExpN}(expn))); & expn \in TExpN \\ ain_{AExpN}(expn); & expn \in AExpN \end{cases}
\end{aligned}$$

Further, the expression that is represented by an arbitrary expression node is given by

$$\begin{aligned} exp_{ExpN} : ExpN &\mapsto Exp \\ expn &\mapsto \begin{cases} exp_{MExpN}(expn); expn \in MExpN \\ exp_{TExpN}(expn); expn \in TExpN \\ exp_{AExpN}(expn); expn \in AExpN \end{cases} \end{aligned}$$

There are three kinds of expressions which are attached to transitions: trigger expressions, guard expressions, and action expressions. There is up to one trigger expression per transition, which is represented by an expression node and attached to the corresponding transition node:

$$\begin{aligned} trg_{TN_{spec}} : TN_{spec} &\rightarrow \mathcal{P}(TrgExpN) \\ (min, trans) &\mapsto \{((min, trans), exp, trg) \mid exp \in sig_T(trans)\} \end{aligned}$$

For each transition, up to one boolean guard expression exists. If there is none, this is interpreted as the expression  $exp_{true}$  that always evaluates to true. A corresponding expression node is created:

$$\begin{aligned} grd_{TN_{spec}} : TN_{spec} &\rightarrow \mathcal{P}(GrdExpN) \\ (min, trans) &\mapsto \{((min, trans), exp, grd) \mid exp \in grd_T(trans)\} \\ &; grd_T(trans) \neq \emptyset \\ (min, trans) &\mapsto \{((min, trans), exp_{true}, grd)\} \\ &; grd_T(trans) = \emptyset \end{aligned}$$

The sequence of actions of a transition defines a sequence of action nodes within the context of the transition node:

$$\begin{aligned} act_{TN_{spec}} : TN_{spec} &\rightarrow seq ActExpN \\ (min, trans) &\mapsto san \\ &; san = \langle an_1, \dots, an_k \rangle \wedge k = |act_T(trans)| \wedge \\ &\quad \forall i \in \{1..k\} \bullet an_i = ((min, trans), act_T(trans)(i), act) \end{aligned}$$

Expressions which are attached to modes are distinguished as flow expressions and invariant expressions. Note that there is no technical difference between the specification's flow and algebraic constraints; they are subsumed as flow expressions here:

$$\begin{aligned} flow_{MIN_{spec}} : MIN_{spec} &\rightarrow \mathcal{P}(FlowExpN) \\ (sm, mi, am) &\mapsto \{((sm, mi, am), exp, flow) \mid \\ &\quad exp \in flow_M(mode_{MI}(mi)) \vee exp \in alge_M(mode_{MI}(mi))\} \end{aligned}$$

Invariant expressions are represented by corresponding nodes:

$$\begin{aligned} inv_{MIN_{spec}} : MIN_{spec} &\rightarrow \mathcal{P}(InvExpN) \\ (sm, mi, am) &\mapsto \{((sm, mi, am), exp, inv) \mid exp \in inv_M(mode_{MI}(mi))\} \end{aligned}$$

Init state constraint expression nodes are constructed from the available agent instance nodes and the init state constraint expressions of the corresponding

agent instances and agents:

$$\begin{aligned} isc_{AIN_{spec}} : AIN_{spec} &\rightarrow \mathcal{P}(IscExpN) \\ (sa, ai, n) &\mapsto \{((sa, ai, n), exp, isc) \mid \\ &exp \in initState_{AI}(ai) \cup initState_A(agent_{AI}(ai))\} \end{aligned}$$

All init state constraint expression nodes are collected recursively for agent instance nodes, such that each agent instance node is mapped to its own init states and the init states of its ancestor nodes:

$$\begin{aligned} allisc_{AIN_{spec}} : AIN_{spec} &\rightarrow \mathcal{P}(IscExpN) \\ (\langle p \rangle \hat{\ } sq, ai, n) &\mapsto isc_{AIN_{spec}}(\langle p \rangle \hat{\ } sq, ai, n) \cup allisc_{AIN_{spec}}(p) \\ (\langle \rangle, ai, n) &\mapsto isc_{AIN_{spec}}(\langle \rangle, ai, n) \end{aligned}$$

The sets of available expression nodes for a given HybridUML specification are:

$$\begin{aligned} ExpN_{spec} &= TrgExpN_{spec} \cup GrdExpN_{spec} \cup ActExpN_{spec} \cup FlowExpN_{spec} \\ &\cup InvExpN_{spec} \\ TrgExpN_{spec} &= \{txn \in TrgExpN \mid \exists tn \in TN_{spec} \bullet txn \in trg_{TN_{spec}}(tn)\} \\ GrdExpN_{spec} &= \{gxn \in GrdExpN \mid \exists tn \in TN_{spec} \bullet gxn \in grd_{TN_{spec}}(tn)\} \\ ActExpN_{spec} &= \{axn \in ActExpN \mid \exists tn \in TN_{spec} \bullet axn \in ran(act_{TN_{spec}}(tn))\} \\ FlowExpN_{spec} &= \\ &\{fxn \in FlowExpN \mid \exists min \in MIN_{spec} \bullet fxn \in flow_{MIN_{spec}}(min)\} \\ InvExpN_{spec} &= \{ixn \in InvExpN \mid \exists min \in MIN_{spec} \bullet ixn \in inv_{MIN_{spec}}(min)\} \\ IscExpN_{spec} &= \{ixn \in IscExpN \mid \exists ain \in AIN_{spec} \bullet ixn \in isc_{AIN_{spec}}(ain)\} \\ TExpN_{spec} &= TrgExpN_{spec} \cup GrdExpN_{spec} \cup ActExpN_{spec} \\ MExpN_{spec} &= FlowExpN_{spec} \cup InvExpN_{spec} \\ AExpN_{spec} &= IscExpN_{spec} \end{aligned}$$

## 5.2 Evaluation Semantics of HybEL Expressions

In this section, the semantics for the evaluation of HybEL expressions is appended. For its definition, the notion of agent instance nodes (from section 5.1) is needed. It is needed to define values for (1) multiplicity expressions of agent instances, properties, and signals, as well as for (2) index specifications of properties and signals.

The semantics given here extends the skeleton semantics of section 3.4, in that it defines values for properties that are equipped with a value specification. Further, the evaluation of sets of integers is added, as it is needed for the index specifications of properties and signals.

**Expression Context of Agents.** For the evaluation of expressions, the variable and signal context is required. This is defined by the agent which contains the expression, such that exactly the agent's variables and signals are available:

$$\begin{aligned} ctx_A : A &\rightarrow CTX \\ a &\mapsto (var_A(a), \{(s, ac) \mid s \in sig_A(a) \wedge \exists p \in port_{A, sig}(a) \bullet acc_{P_s}(p) = ac\}) \end{aligned}$$

**Expression Context of Agent Instance Nodes.** For convenience, the context given by agent instance nodes is defined in a straight-forward way, such that it is given by the agent instance node's agent:

$$\begin{aligned} ctx_{AIN} &: AIN \rightarrow CTX \\ ain &\mapsto ctx_A(agent_{AI}(ai_{AIN}(ain))) \end{aligned}$$

**Semantics of Expressions.** The *evaluation semantics* of hybel expressions is then given by

$$\begin{aligned} eval &: Exp \times AIN \rightarrow VAL_{eval} \cup \{\lambda\} \\ eval(exp, ain) &= eval_{ht}(ht(ctx_{AIN}(ain))(exp), ain) \end{aligned}$$

Each hybel item tree along with an agent instance node is mapped to either a value, or to the special value  $\lambda$  that denotes the absence of a value:

$$eval_{ht} : tree_o HybelItem \times AIN \rightarrow VAL_{eval} \cup \{\lambda\}$$

Literal values are the values (as for  $eval_{ht, \emptyset}$ ):

$$\begin{aligned} (((t, lit), l), sub), ain) &\mapsto l; t \in DT \\ (((sdt_{anon}, lit), v), \langle t_1, \dots, t_n \rangle), ain) \\ &\mapsto \langle eval_{ht}(t_1, ain), \dots, eval_{ht}(t_n, ain) \rangle \end{aligned}$$

Operations are evaluated (as for  $eval_{ht, \emptyset}$ ):

$$\begin{aligned} (((t, op), \diamond), \langle t_1, t_2 \rangle), ain) &\mapsto eval_{ht}(t_1, ain) \diamond eval_{ht}(t_2, ain) \\ ; t_1, t_2 &\in \{int, real, anaReal\} \wedge \diamond \in \{+, -, \cdot, /, ^, <, \leq, \geq, >\} \\ \vee t_1 = t_2 &\wedge \diamond \in \{=, \neq\} \\ \vee t_1, t_2 &\in \{bool\} \wedge \diamond \in \{\wedge, \vee\} \\ (((t, op), \neg), \langle t_1 \rangle), ain) &\mapsto \neg eval_{ht}(t_1, ain) \end{aligned}$$

Integer set specifications define sets of integers:

$$\begin{aligned} (((intSet, intSpecs), val), \langle \\ &(((intSet, intRange), val_1), \langle s_{low,1}, s_{up,1} \rangle), \dots, \\ &(((intSet, intRange), val_n), \langle s_{low,n}, s_{up,n} \rangle) \rangle), ain) \\ &\mapsto \{eval_{ht}(s_{low,1}, ain), \dots, eval_{ht}(s_{up,1}, ain)\} \cup \dots \cup \\ &\{eval_{ht}(s_{low,n}, ain), \dots, eval_{ht}(s_{up,n}, ain)\} \end{aligned}$$

Variables are mapped to assigned values, if the agent instance node has a corresponding value specification. If an index sub-expression is available, it is evaluated itself, otherwise index 0 is applied:

$$\begin{aligned} (((t, var), ((t, (v, acc)), \langle (indexExp, htree), sub_{id} \rangle)), sub), ain) \\ &\mapsto \sigma_{VN_{spec}}(ain, v, p, pi, eval_{ht}(htree, ain)) \\ &\text{with unique } (ain, v, p, pi, eval_{ht}(htree, ain)) \in vn_{AIN}(tree_{AIN}(ain)) \\ (((t, var), ((t, (v, acc)), \langle \rangle)), sub), ain) &\mapsto \sigma_{VN_{spec}}(ain, v, p, pi, 0) \\ &\text{with unique } (ain, v, p, pi, 0) \in vn_{AIN}(tree_{AIN}(ain)) \end{aligned}$$

The remaining expressions have no evaluation result:

$$(t, ain) \mapsto \lambda; \text{ else}$$

### 5.3 Prerequisites for Code Creation

This section provides some prerequisites that are used for code creation (discussed in section 5.5), for the definition of local HL<sup>3</sup> variables, which is done in section 5.4, as well as for the HybridUML-specific definition of operations of abstract subjects in section 5.6.

First, the determination of different sets of variables and signals from expressions is defined, followed by the determination of variable and signal nodes from expression nodes. Finally, the hybel item tree representation of expressions from expression nodes is given.

#### 5.3.1 Variables and Signals of Expressions

As a prerequisite for the mapping of expression nodes to programs, the sets of HybridUML variables and signals, which are accessed by particular expressions, are determined.

**Read Variables of Hybel Trees.** The variables which are read by an expression are given by a mapping of hybel item trees to sets of HybridUML variables:

$$var_{ht,read} : tree_o \text{ HybelItem} \rightarrow \mathcal{P}(V)$$

For a hybel item tree that represents a variable, the variable itself is contained. Additionally, variables of subexpressions (for structured data types) as well as variables contained in the identifier item tree (as part of the index expression) are collected. The corresponding definition of  $var_{it,read}$  is given below.

$$\begin{aligned} &(((t, r), ((t_{id}, (v, acc)), sub_{id})), \langle s_1, \dots, s_n \rangle) \mapsto \\ &\quad \{v\} \cup \bigcup_{i=1}^n var_{ht,read}(s_i) \cup var_{it,read}(((t_{id}, (v, acc)), sub_{id})) \\ &\text{with } r \in \{var, derivVar\} \end{aligned}$$

Expressions that send signals may contain read variables in subexpressions, from which values of signal parameters are created. Further, a signal's index expression can contain read variables, too:

$$\begin{aligned} &(((sigtype, sendSig), idtree), \langle s_1, \dots, s_n \rangle) \mapsto \\ &\quad \bigcup_{i=1}^n var_{ht,read}(s_i) \cup var_{it,read}(idtree) \end{aligned}$$

For trigger expressions, i.e. expressions that read signals, only variables of the index expression are collected (via the identifier item tree). Variables of subexpressions are *written* and thus left out.

$$(((sigtype, recvSig), idtree), sub) \mapsto var_{it,read}(idtree)$$

Most assignment expressions contain a left-hand side that consists of a variable that is written only, and a right-hand side that is read only. Therefore, read variables are collected from the hybel item tree's subexpressions, excluding the first one (which represents the left-hand side). Additionally, the identifier item

tree of the first subexpression is searched for contained index expressions, which again may contain read variables:

$$\begin{aligned} &(((t, r), val), \langle ((t_1, var), idtree_1), sub_1, s_2, \dots, s_n \rangle) \mapsto \\ &\quad \bigcup_{i=2}^n var_{ht,read}(s_i) \cup var_{it,read}(idtree_1) \\ &\quad \text{with } r \in \{ass, intNondetAss, diffAss\} \end{aligned}$$

Note that assignments *to* derivatives are excluded above, since the derivative's variable is not only written, but also read. Therefore, for those assignments, the first subexpression is included in the generic rule given below.

Index assignment expressions are treated separately, too, because they explicitly do not contain any read variables:

$$(((int, indexAss), val), sub) \mapsto \emptyset$$

All remaining hybrid item trees are mapped according to the following generic rule, that does not collect variables from the expression itself, but from all contained subexpressions:

$$(hi, \langle s_1, \dots, s_n \rangle) \mapsto \bigcup_{i=1}^n var_{ht,read}(s_i); \text{ else}$$

**Read Variables of Identifier Trees.** The variables which are read by an identifier expression are given by the mapping of identifier item trees to sets of HybridUML variables:

$$var_{it,read} : tree_o IdItem \rightarrow \mathcal{P}(V)$$

Since the identifier itself is handled by the mapping  $var_{ht,read}$ , this mapping only collects contained variables of index expressions:

$$((indexExp, htree), sub) \mapsto var_{ht,read}(htree)$$

For subexpressions, representing sub-variables of a structured data type, also only index expressions are searched; therefore the sub-variables themselves are omitted. This is desired, because variables of structured data type will always be read or written entirely within the generated code.

$$((t, val), \langle s_1, \dots, s_n \rangle) \mapsto \bigcup_{i=1}^n var_{it,read}(s_i); \text{ else}$$

**Derivative Read Access of Hybel Trees.** The variables for which the derivative value is read are collected for each hybrid item tree, by the function

$$var_{ht,readDeriv} : tree_o HybelItem \rightarrow \mathcal{P}(V)$$

Trees that represent derivatives map to their variables:

$$(((anaReal, derivVar), ((anaReal, (v, acc)), sub_{id})), \langle s_1, \dots, s_n \rangle) \mapsto \{v\}$$

From the above variables, the ones that are only written shall be excluded; therefore from assignment expressions, the left-hand side is omitted:

$$(((anaReal, diffAss), val), \langle s_1, \dots, s_n \rangle) \mapsto \bigcup_{i=2}^n var_{ht, readDeriv}(s_i)$$

All remaining trees are only descended, such that their sub-expressions are searched for respective variables:

$$(item, \langle s_1, \dots, s_n \rangle) \mapsto \bigcup_{i=1}^n var_{ht, readDeriv}(s_i); \text{ else}$$

**Previous Value Access of Hybel Trees.** In contrast to “conventional” read access to variables, some expressions require the read access to the previous value of a variable. Those variables are given by the mapping

$$var_{ht, readPrev} : tree_o HybelItem \rightarrow \mathcal{P}(V)$$

Variables for which the derivative value is read within the expression are included, in the same manner as for  $var_{ht, readDeriv}$ :

$$(((anaReal, derivVar), ((anaReal, (v, acc)), sub_{id})), \langle s_1, \dots, s_n \rangle) \mapsto \{v\}$$

The ones that are only written shall be excluded, too:

$$(((anaReal, diffAss), val), \langle s_1, \dots, s_n \rangle) \mapsto \bigcup_{i=2}^n var_{ht, readPrev}(s_i)$$

Additionally, real-valued variables which take part in an equality comparison are included. This is necessary, because of the discretization of continuous steps, an approximation has to be done for those comparisons. See section 5.5 for details.

$$\begin{aligned} &(((bool, op), val), (((role_1, val_1), sub_1), ((role_2, val_2), sub_2))) \mapsto \\ &var_{ht, read}(((role_1, val_1), sub_1)) \cup var_{ht, read}(((role_2, val_2), sub_2)) \\ &\text{with } \{role_1, role_2\} \cap \{(real, op), (anaReal, diffOp), (real, lit), \\ &(real, var), (anaReal, var), (anaReal, derivVar)\} \\ &\neq \emptyset \end{aligned}$$

All remaining trees are only descended, such that their sub-expressions are searched for respective variables:

$$(item, \langle s_1, \dots, s_n \rangle) \mapsto \bigcup_{i=1}^n var_{ht, readPrev}(s_i); \text{ else}$$

**Read Signals of Hybel Trees.** Read signals, i.e. triggers are only given directly by a trigger expression. Since there are no nested triggers, the complete mapping is

$$\begin{aligned} sig_{ht, read} : tree_o HybelItem &\rightarrow \mathcal{P}(S) \\ &(((sigtype, recvSig), ((t_{id}, (s, acc)), sub_{id})), sub) \mapsto \{s\} \\ hi &\mapsto \emptyset; \text{ else} \end{aligned}$$

**Written Variables of Hybel Trees.** The variables which are written by an expression are given by a corresponding mapping of hybel item trees to sets of HybridUML variables:

$$var_{ht,write} : tree_o HybelItem \rightarrow \mathcal{P}(V)$$

For any kind of assignment statement, the first subexpression represents a variable that is written:

$$\begin{aligned} &(((t, r), val), \langle (((t_1, r_1), ((t_{1,id}, (v, acc)), sub_{1,id})), sub_1), s_2, \dots, s_n) \rangle) \mapsto \\ & \quad \{v\} \cup \bigcup_{i=2}^n var_{ht,write}(s_i) \\ & \text{with } r \in \{ass, diffAss, intNondetAss, indexAss\} \\ & \quad \wedge r_1 \in \{var, derivVar\} \end{aligned}$$

Trigger expressions can contain parameter subexpressions, that consist of variables that are assigned with the actual parameter values. Additionally, the signal may have an attached index assignment expression, included in the identifier item tree:

$$\begin{aligned} &(((sigtype, recvSig), idtree), \langle \\ & \quad (((t_1, r_1), ((t_{1,id}, (v_1, acc)), sub_{1,id})), sub_1), \dots, \\ & \quad (((t_n, r_n), ((t_{n,id}, (v_n, acc)), sub_{n,id})), sub_n) \rangle) \\ & \mapsto var_{it,write}(idtree) \cup \{v_1, \dots, v_n\} \end{aligned}$$

For all other trees, the variables of their sub-expressions are collected:

$$(item, \langle s_1, \dots, s_n \rangle) \mapsto \bigcup_{i=1}^n var_{ht,write}(s_i); \text{ else}$$

**Written Variables of Identifier Trees.** The variables which are written by an identifier expression are given by

$$var_{it,write} : tree_o IdItem \rightarrow \mathcal{P}(V)$$

Since the identifier itself is handled by the mapping  $var_{ht,write}$ , this mapping only collects contained variables of index expressions:

$$((indexExp, htree), sub) \mapsto var_{ht,write}(htree)$$

Like for  $var_{it,read}$ , sub-variables of a structured data type are omitted. Therefore the recursive descending into the identifier tree only affects index expressions:

$$((t, val), \langle s_1, \dots, s_n \rangle) \mapsto \bigcup_{i=1}^n var_{it,write}(s_i); \text{ else}$$

**Derivative Write Access of Hybel Trees.** The variables for which the derivative value is written are collected for each hybel item tree:

$$var_{ht,writeDeriv} : tree_o HybelItem \rightarrow \mathcal{P}(V)$$

The variable of the left-hand side of an assignment expression is contained, if it is a derivative:

$$\begin{aligned} &(((anaReal, diffAss), val), \\ & \quad \langle (((anaReal, derivVar), ((anaReal, (v, acc)), sub_{id})), sub), s_2, \dots, s_n \rangle) \\ & \mapsto \{v\} \end{aligned}$$

For other trees, the sub-expressions are searched for respective variables:

$$(item, \langle s_1, \dots, s_n \rangle) \mapsto \bigcup_{i=1}^n var_{ht, writeDeriv}(s_i); \text{ else}$$

**Written Signals of Hybel Trees.** Written signals are given by the mapping

$$sig_{ht, write} : tree_o HybelItem \rightarrow \mathcal{P}(S)$$

A signal raise statement contains a written signal:

$$(((sigtype, sendSig), ((t_{id}, (s, acc)), sub_{id})), sub) \mapsto \{s\}$$

Some other kinds of expressions may host a signal raise statement, therefore all remaining expressions are descended:

$$(item, \langle s_1, \dots, s_n \rangle) \mapsto \bigcup_{i=1}^n sig_{ht, write}(s_i); \text{ else}$$

### 5.3.2 Variable and Signal Nodes of Expression Nodes

For HybridUML variables and signals, there are mappings that provide the variable or signal nodes which represent the given variable or signal within the context of an expression node. Each variable or signal node represents a dedicated index of the respective variable or signal, therefore the number of nodes and the multiplicity of the variable or signal are equal.

$$\begin{aligned} vn_{expn} : V \times ExpN &\rightarrow \mathcal{P}(VN) \\ (v, expn) &\mapsto \{vn \in vn_{AIN}(tree_{AIN}(ain_{ExpN}(expn))) \mid var_{VN}(vn) = v\} \\ sn_{expn} : S \times ExpN &\rightarrow \mathcal{P}(SN) \\ (s, expn) &\mapsto \{sn \in sn_{AIN}(tree_{AIN}(ain_{ExpN}(expn))) \mid sig_{SN}(sn) = s\} \end{aligned}$$

### 5.3.3 Hybel Item Trees of Expression Nodes

For convenience, a mapping from expression nodes to the hybel item tree that represents the expression of the node is defined. For this, the mapping from agent instance nodes to expression contexts of section 5.2 is used:

$$\begin{aligned} ht_{expn} : ExpN_{spec} &\rightarrow tree_o HybelItem \\ expn &\mapsto ht(ctx_{AIN}(ain_{ExpN}(expn)))(exp_{ExpN}(expn)) \end{aligned}$$

## 5.4 HL<sup>3</sup> Model Definition

Based on the intermediate representation of HybridUML specifications given in section 5.1, we instantiate the HL<sup>3</sup> model that represents a particular HybridUML specification in this section. Therefore, we define the corresponding constant part  $c_{spec} \in CONST$  of the state space of the operational HL<sup>3</sup> semantics, as it is predetermined and discussed in section 4.2.

Below, we follow the structure of section 4.2, such that all components of  $CONST$  are defined here in the same order as they appear in section 4.2. As an example for the resulting HL<sup>3</sup> model, a C++ variant of  $c_{tc} \in CONST$  of the case study “Radio-Based Train Control” is presented in appendix D.1, created by the implementation of transformation  $\Phi_{HUML}$ .

The definition of code creation from HyBEL expressions, which is particularly needed for the definition of programs for program subjects, will be given in section 5.5.

Finally, the definition of HybridUML-specific behavior for abstract subjects follows in section 5.6.

### 5.4.1 Entities

**scheduler.** This is the pre-defined HL<sup>3</sup> scheduler, as for every HL<sup>3</sup> model:

$$sched(c_{spec}) = scheduler$$

**SELECTOR.** The applied selector is the pre-defined HybridUML selector:

$$sel(c_{spec}) = selector_{HUML}$$

**AM.** The abstract machines of the system represent the sequential behavioral components, which in turn are defined by basic agent instance nodes  $ain \in AIN_{basic}$ . Therefore, for each  $ain$ , an abstract machine is created, i.e. there is a bijection

$$am_{HL3, AIN_{basic}} : AIN_{basic} \xrightarrow{\sim} Am$$

that defines the abstract machines of the HL<sup>3</sup> model:

$$am(c_{spec}) = Am$$

The inverse function that maps abstract machines back to agent instance nodes is given as  $ain_{Am} = am_{HL3, AIN_{basic}}^{-1}$ .

**IFM.** For a HybridUML simulation, there are no interface modules:

$$ifm(c_{spec}) = \emptyset$$

**FLOW.** The flows of the HL<sup>3</sup> model are given by the flow expression nodes. Each  $fxn \in FlowExpN_{spec}$  defines an HL<sup>3</sup> flow, i.e. there is a bijection (and the corresponding inverse)

$$\begin{aligned} flow_{HL3, FlowExpN_{spec}} : FlowExpN_{spec} &\xrightarrow{\sim} Flow \\ fxn_{Flow} &= flow_{HL3, FlowExpN_{spec}}^{-1} \end{aligned}$$

This defines the flows of the HL<sup>3</sup> model:

$$flow(c_{spec}) = Flow$$

**TRANS.** The transitions of the HL<sup>3</sup> model are defined by the transition nodes, in that for each transition node a unique transition is generated:

$$\begin{aligned} trans_{HL3, TN_{spec}} &: TN_{spec} \rightsquigarrow Trans \\ tn_{trans} &= trans_{HL3, TN_{spec}}^{-1} \end{aligned}$$

Those are the transitions of the model:

$$trans(c_{spec}) = Trans$$

**VAR.** The local variables  $var(c_{spec})$  are defined for the expression nodes  $ExpN_{spec}$ , such that they can be used by their generated programs. There are two kinds of variables – (1) *simple variables* and (2) *array variables*. All variables that represent HybridUML variables or signals are array variables, such that according to the multiplicity defined by the HybridUML variable or signal, the local HL<sup>3</sup> variable contains a corresponding number of *subscripted variables*. In contrast, most of the additionally introduced HL<sup>3</sup> variables are simple variables.

In the following, several mappings are given that define the available local HL<sup>3</sup> variables. For every variable, its type is defined, too. The union of all types  $Val$  contains all HybridUML-specific data values:

$$VAL_{eval} \subseteq Val$$

*HybridUML Properties.* Every HybridUML property  $v \in V \cup V_{local}$  (including bound variables of quantified expressions) that is read or written by the expression of an expression node is represented by a local variable. For the calculation of flow integration steps, a previous value may be needed:

$$lvar_{V, V_{local}} : ExpN \times (V \cup V_{local}) \times \{read, write\} \times \{cur, prev\} \leftrightarrow Var$$

Since  $lvar_{V, V_{local}}$  is a partial function, local variables are not generated for every tuple of the domain; precisely, local variables are created for each variable of expression nodes from the given HybridUML specification, (1) for which the current value is read, (2) for which the previous value is read, or (3) for which a (current) value is written:

$$\begin{aligned} \text{dom } lvar_{V, V_{local}} &= \\ &\{(expn, v, read, cur) \mid expn \in ExpN_{spec} \wedge v \in var_{ht, read}(ht_{expn}(expn))\} \\ &\cup \{(expn, v, read, prev) \\ &\quad \mid expn \in ExpN_{spec} \wedge v \in var_{ht, readPrev}(ht_{expn}(expn))\} \\ &\cup \{(expn, v, write, cur) \mid expn \in ExpN_{spec} \wedge v \in var_{ht, write}(ht_{expn}(expn))\} \end{aligned}$$

The mapping  $lvar_{V, V_{local}}$  is not injective, because for some expression nodes, the local variables that represent the same HybridUML variable coincide: As defined later, the sequence of actions of a transition node, along with its trigger, define a single shared program. Within, any sequence of read and write accesses

to the same HybridUML variable  $v_{hyuml}$  may occur via local variables  $v_1, \dots, v_n$  for expression nodes  $expn_1, \dots, expn_n$ . In order to ensure that every write access is effective for subsequent read accesses, exactly those variables are identical:

$$\begin{aligned} & \forall (expn_1, v_1, acc_1, tm_1), (expn_2, v_2, acc_2, tm_2) \in \text{dom } lvar_{V, V_{local}} \bullet \\ & \quad \{expn_1, expn_2\} \subseteq TExpN \wedge tn_{TExpN}(expn_1) = tn_{TExpN}(expn_2) \\ & \quad \wedge v_1 = v_2 \wedge tm_1 = tm_2 = cur \\ & \Leftrightarrow \\ & \quad lvar_{V, V_{local}}(expn_1, v_1, acc_1, tm_1) = lvar_{V, V_{local}}(expn_2, v_2, acc_2, tm_2) \end{aligned}$$

Note that also for a single expression node, this means that read and write access are realized with the same local variable.

The types of the created variables are defined by the types of the corresponding HybridUML variable:

$$\begin{aligned} & \forall (expn, v, acc, tm) \in \text{dom } lvar_{V, V_{local}} \bullet \\ & \quad type_{Var}(lvar_{V, V_{local}}(expn, v, acc, tm)) = \begin{cases} type_{HL3, V}(v) & ; v \in V \\ \mathbb{N}_0 \rightarrow \mathbb{Z} & ; v \in V_{local} \end{cases} \end{aligned}$$

The types are defined for HybridUML variables and HybridUML data types. Since HybridUML variables have multiplicities, the valuation of a corresponding HL<sup>3</sup> variable is given as a function of natural numbers to the type of the subscripted variables. The array variable's type is then the set of these functions. Formally, there is no upper bound for array indices of this array representation, but for an implementation, an upper bound defined by the multiplicity would be used explicitly.

$$\begin{aligned} & type_{HL3, V} : V \rightarrow \mathcal{P}(Val) \\ & \quad v \mapsto (\mathbb{N}_0 \rightarrow type_{HL3, DT}(type_V(v))) \\ & type_{HL3, DT} : DT \rightarrow \mathcal{P}(Val) \\ & \quad bool \mapsto \mathbb{B} \\ & \quad int \mapsto \mathbb{Z} \\ & \quad real \mapsto \mathbb{R} \\ & \quad anaReal \mapsto \mathbb{R} \\ & \quad et \mapsto lit_{DT}(et); et \in DT_{enum} \\ & \quad sdt \mapsto \{ \langle type_{HL3, DT}(varseq_{DT}(sdt)(1)), \dots, \\ & \quad \quad type_{HL3, DT}(varseq_{DT}(sdt)(n)) \rangle \mid |varseq_{DT}(sdt)| = n \} \\ & \quad ; sdt \in DT_{struc} \end{aligned}$$

Note that bound variables from HyBEL expressions are also of array type, such that the handling of them and HybridUML variables is the same. However, these are effectively simple variables, because there is only one subscripted variable inside.

*HybridUML Signals.* Every HybridUML signal  $s \in S$  that is received or sent by the expression of a node is represented by a variable:

$$lvar_S : ExpN \times S \times \{read, write\} \rightsquigarrow Var$$

The variables are of boolean type, such that the value *true* denotes that the signal is currently active, within the scope of the corresponding program. Programs can either read this value, such that for trigger expressions, the enabledness of the corresponding transition is affected, or write this value. This will be done for signal raise statements, in order to activate a signal for the complete system, or for trigger statements, in order to consume the signal locally.

$$\forall (expn, s, acc) \in \text{dom } lvar_S \bullet \text{type}_{Var}(lvar_S(expn, s, acc)) = \mathbb{B}$$

Local variables are created for the combinations of signals and expression nodes that actually use them:

$$\begin{aligned} \text{dom } lvar_S = & \\ & \{(expn, s, read) \mid expn \in \text{ExpN}_{spec} \wedge s \in \text{sig}_{ht,read}(ht_{expn}(expn))\} \\ & \cup \{(expn, s, write) \mid expn \in \text{ExpN}_{spec} \wedge v \in \text{sig}_{ht,write}(ht_{expn}(expn))\} \end{aligned}$$

*Parameters of HybridUML Signals.* For every HybridUML signal, there are up to two local variables for each parameter of the signal within the corresponding expression nodes, one variable for the reception and one for the sending of the parameter:

$$lvar_{SigParam} : \text{ExpN} \times S \times \{read, write\} \times \mathbb{N}_0 \rightsquigarrow Var$$

For the combinations of signals and their parameter indices and expression nodes that actually use the signals, local variables are created:

$$\begin{aligned} \text{dom } lvar_{SigParam} = & \\ & \{(expn, s, read, n) \mid expn \in \text{ExpN}_{spec} \wedge s \in \text{sig}_{ht,read}(ht_{expn}(expn)) \\ & \quad \wedge 0 \leq n < |sn_{expn}(s, expn)|\} \\ & \cup \{(expn, s, write) \mid expn \in \text{ExpN}_{spec} \wedge v \in \text{sig}_{ht,write}(ht_{expn}(expn)) \\ & \quad \wedge 0 \leq n < |sn_{expn}(s, expn)|\} \end{aligned}$$

The parameter's type is defined by its the HybridUML data type, as defined for HybridUML variables above:

$$\begin{aligned} \forall (expn, s, acc, n) \in \text{dom } lvar_{SigParam} \bullet \text{type}_{Var}(lvar_{SigParam}(expn, s, acc, n)) \\ = \text{type}_{HL3,DT}(\text{paramTypes}_S(s)(n + 1)) \end{aligned}$$

*Index Counters.* For the reception of signals, a local *trigger index* variable is created that stores the signal index wrt. its multiplicity for which the signal's occurrence is received:

$$lvar_{trgIdx} : \text{ExpN} \times S \rightsquigarrow Var$$

This is done for each trigger within a corresponding expression node:

$$\begin{aligned} \text{dom } lvar_{trgIdx} = & \\ & \{(expn, s) \mid expn \in \text{ExpN}_{spec} \wedge s \in \text{sig}_{ht,read}(ht_{expn}(expn))\} \end{aligned}$$

A local *signal index* exists for the sending of signals:

$$lvar_{sigIdx} : \text{ExpN} \times S \rightsquigarrow Var$$

A corresponding variable is generated for sent signals within a corresponding expression node:

$$\begin{aligned} \text{dom } lvar_{sigIdx} = \\ \{(expn, s) \mid expn \in ExpN_{spec} \wedge s \in sig_{ht,write}(ht_{expn}(expn))\} \end{aligned}$$

Index variables are of natural number type:

$$\begin{aligned} \forall (expn, s) \in \text{dom } lvar_{trgIdx} \bullet type_{Var}(lvar_{trgIdx}(expn, s)) = \mathbb{N}_0 \\ \forall (expn, s) \in \text{dom } lvar_{sigIdx} \bullet type_{Var}(lvar_{sigIdx}(expn, s)) = \mathbb{N}_0 \end{aligned}$$

*Derivative Helper Variables.* From HybridUML variables that are used to access derivative values, local variables to store the publication time ticks of the current value and of the previous value are generated, as well as local variables to store the time difference.

$$\begin{aligned} lvar_{V,\Delta t} : ExpN \times V \rightsquigarrow Var \\ lvar_{V,curTck} : ExpN \times V \rightsquigarrow Var \\ lvar_{V,prevTck} : ExpN \times V \rightsquigarrow Var \end{aligned}$$

For combinations of read derivatives within expression nodes, variables for the previous time stamp and the difference exist:

$$\begin{aligned} \text{dom } lvar_{V,prevTck} = \text{dom } lvar_{V,\Delta t} = \\ \{(expn, v) \mid expn \in ExpN_{spec} \wedge v \in var_{ht,readDeriv}(ht_{expn}(expn))\} \end{aligned}$$

The current time tick is also used for written derivatives:

$$\begin{aligned} \text{dom } lvar_{V,curTck} = \\ \{(expn, v) \mid expn \in ExpN_{spec} \wedge v \in var_{ht,readDeriv}(ht_{expn}(expn)) \\ \cup var_{ht,writeDeriv}(ht_{expn}(expn))\} \end{aligned}$$

The time tick variables hold *ModelTime* values, the time difference is a natural number:

$$\begin{aligned} \forall (expn, v) \in \text{dom } lvar_{V,curTck} \bullet \\ type_{Var}(lvar_{V,curTck}(expn, v)) = ModelTime \\ \forall (expn, v) \in \text{dom } lvar_{V,prevTck} \bullet \\ type_{Var}(lvar_{V,prevTck}(expn, v)) = ModelTime \\ \forall (expn, v) \in \text{dom } lvar_{V,\Delta t} \bullet \\ type_{Var}(lvar_{V,\Delta t}(expn, v)) = \mathbb{N}_0 \end{aligned}$$

*Quantified Expression Results.* The result of quantified boolean expressions is stored in an exclusive local variable:

$$lvar_{quantRes} : ExpN \times tree_o HybelItem \rightsquigarrow Var$$

Therefore, a variable exists for each occurrence of a quantified expression in an expression node:

$$\begin{aligned} \text{dom } lvar_{quantRes} = \\ \{(expn, (((bool, q), val), sub)) \mid expn \in ExpN_{spec} \wedge \\ (((bool, q), val), sub) \in subtrees_{o,HybelItem}(ht_{expn}(expn)) \wedge q \in \{\forall, \exists\}\} \end{aligned}$$

The result variable is of boolean type:

$$\forall (expn, htree) \in \text{dom } lvar_{quantRes} \bullet \text{type}_{Var}(lvar_{quantRes}(expn, htree)) = \mathbb{B}$$

*Visibility Set Helper Variables.* A visibility set  $v \in \text{VisibilitySet}$  is provided for the publication of written values. It gets the respective visibility set parameter for HL<sup>3</sup> flows and HL<sup>3</sup> transitions by default, but can be re-calculated for specific purposes:

$$lvar_{visSet} : \text{ExpN} \rightarrow \text{Var}$$

A corresponding variable exists for flow expression nodes and for action and trigger expression nodes. The variables for actions and triggers coincide for common transitions:

$$\begin{aligned} \text{dom } lvar_{visSet} &= \text{TrgExpN}_{spec} \cup \text{ActExpN}_{spec} \cup \text{FlowExpN}_{spec} \\ \forall expn_1, expn_2 \in \text{dom } lvar_{visSet} \bullet \\ & lvar_{visSet}(expn_1) = lvar_{visSet}(expn_2) \Leftrightarrow \\ & \{expn_1, expn_2\} \subseteq \text{TrgExpN}_{spec} \cup \text{ActExpN}_{spec} \\ & \wedge tn_{TExpN}(expn_1) = tn_{TExpN}(expn_2) \\ & \vee expn_1 = expn_2 \end{aligned}$$

For the calculation of visibility sets, several local helper variables are needed. They are used in the context of the assignment of a derivative variable, and for resetting a received signal on its consumption. (1) A visibility variable stores a single visibility  $vis \in \text{Visibility}$ . It is only used for expression nodes that either assign a derivative or reset a signal:

$$\begin{aligned} lvar_{vis} &: \text{ExpN} \rightarrow \text{Var} \\ \text{dom } lvar_{vis} &= \text{dom } lvar_{visSet} \cap (\text{TrgExpN}_{spec} \cup \\ & \{expn \in \text{ExpN}_{spec} \mid \exists v \in V \bullet v \in \text{var}_{ht, writeDeriv}(ht_{expn}(expn))\}) \end{aligned}$$

(2) The publication time tick of the visibility is stored in a local variable:

$$\begin{aligned} lvar_{newTck} &: \text{ExpN} \rightarrow \text{Var} \\ \text{dom } lvar_{newTck} &= \text{dom } lvar_{vis} \end{aligned}$$

(3) For flow integration operations, the entries of the pre-defined visibility set are iterated, by means of a visibility index:

$$\begin{aligned} lvar_{visIdx} &: \text{ExpN} \rightarrow \text{Var} \\ \text{dom } lvar_{visIdx} &= \text{dom } lvar_{visSet} \cap \\ & \{expn \in \text{ExpN}_{spec} \mid \exists v \in V \bullet v \in \text{var}_{ht, writeDeriv}(ht_{expn}(expn))\} \end{aligned}$$

(4) In order to read the pre-defined visibility set for iteration, a local visibility set variable is provided:

$$\begin{aligned} lvar_{visOrig} &: \text{ExpN} \rightarrow \text{Var} \\ \text{dom } lvar_{visOrig} &= \text{dom } lvar_{visIdx} \end{aligned}$$

The types of variables are obvious:

$$\begin{aligned}
&\forall \text{expn} \in \text{dom } \text{lvar}_{\text{visSet}} \bullet \text{type}_{\text{Var}}(\text{lvar}_{\text{visSet}}(\text{expn})) = \text{VisibilitySet} \\
&\forall \text{expn} \in \text{dom } \text{lvar}_{\text{vis}} \bullet \text{type}_{\text{Var}}(\text{lvar}_{\text{vis}}(\text{expn})) = \text{Visibility} \\
&\forall \text{expn} \in \text{dom } \text{lvar}_{\text{visIdx}} \bullet \text{type}_{\text{Var}}(\text{lvar}_{\text{visIdx}}(\text{expn})) = \mathbb{N}_0 \\
&\forall \text{expn} \in \text{dom } \text{lvar}_{\text{newTck}} \bullet \text{type}_{\text{Var}}(\text{lvar}_{\text{newTck}}(\text{expn})) = \text{ModelTime} \\
&\forall \text{expn} \in \text{dom } \text{lvar}_{\text{visOrig}} \bullet \text{type}_{\text{Var}}(\text{lvar}_{\text{visOrig}}(\text{expn})) = \text{VisibilitySet}
\end{aligned}$$

*Non-Deterministic Assignment Helper Variables.* For each non-deterministic assignment of integers, there is a local variable that counts the number of possible values during evaluation, before one is chosen:

$$\text{lvar}_{\text{hitCount}} : \text{ExpN} \times \text{tree}_o \text{HybellItem} \rightsquigarrow \text{Var}$$

Each possible value is stored itself in a corresponding array variable:

$$\text{lvar}_{\text{hits}} : \text{ExpN} \times \text{tree}_o \text{HybellItem} \rightsquigarrow \text{Var}$$

Similarly to quantified expressions, a variable exists for each occurrence of a non-deterministic assignment in an expression node:

$$\begin{aligned}
&\text{dom } \text{lvar}_{\text{hitCount}} = \text{dom } \text{lvar}_{\text{hits}} = \\
&\{( \text{expn}, ((\text{int}, \text{intNondetAss}), \text{val}), \text{sub}) \mid \text{expn} \in \text{ExpN}_{\text{spec}} \wedge \\
&\quad ((\text{int}, \text{intNondetAss}), \text{val}), \text{sub}) \in \text{subtrees}_{o, \text{HybellItem}}(\text{ht}_{\text{expn}}(\text{expn}))\}
\end{aligned}$$

The hit counter holds a natural number, the hit array stores natural-numbered indices:

$$\begin{aligned}
&\forall (\text{expn}, \text{htree}) \in \text{dom } \text{lvar}_{\text{hitCount}} \bullet \text{type}_{\text{Var}}(\text{lvar}_{\text{hitCount}}(\text{expn}, \text{htree})) = \mathbb{N}_0 \\
&\forall (\text{expn}, \text{htree}) \in \text{dom } \text{lvar}_{\text{hits}} \bullet \text{type}_{\text{Var}}(\text{lvar}_{\text{hits}}(\text{expn}, \text{htree})) = (\mathbb{N}_0 \rightarrow \mathbb{N}_0)
\end{aligned}$$

*Boolean Result of Programs.* Programs that represent the calculation of boolean expressions of the HybridUML model have a dedicated *return variable*. The boolean expressions from the model are trigger expressions, guard expressions, and invariant constraints:

$$\begin{aligned}
&\text{BExpN}_{\text{spec}} = \text{TrgExpN}_{\text{spec}} \cup \text{GrdExpN}_{\text{spec}} \cup \text{InvExpN}_{\text{spec}} \cup \text{IscExpN}_{\text{spec}} \\
&\subseteq \text{TExpN} \cup \text{MExpN} \cup \text{AExpN}
\end{aligned}$$

The return variables are identified by the injection

$$\text{retvar}_{\text{BExpN}} : \text{BExpN}_{\text{spec}} \rightsquigarrow \text{Var}$$

They are of boolean type:

$$\forall \text{expn} \in \text{dom } \text{retvar}_{\text{BExpN}} \bullet \text{type}_{\text{Var}}(\text{retvar}_{\text{BExpN}}(\text{expn})) = \mathbb{B}$$

*Local Variables of the HL<sup>3</sup> Model.* The ranges of all the above mappings are disjoint, such that there are unique variables for all described purposes. Then, their union define the set of local variables:

$$\begin{aligned}
var(c_{spec}) = & \\
& \text{ran } lvar_{V, V_{local}} \cup \text{ran } lvar_S \cup \text{ran } lvar_{SigParam} \\
& \cup \text{ran } lvar_{trgIdx} \cup \text{ran } lvar_{sigIdx} \cup \text{ran } lvar_{V, \Delta t} \\
& \cup \text{ran } lvar_{V, curTck} \cup \text{ran } lvar_{V, prevTck} \\
& \cup \text{ran } lvar_{quantRes} \cup \text{ran } lvar_{vis} \cup \text{ran } lvar_{visSet} \\
& \cup \text{ran } lvar_{visIdx} \cup \text{ran } lvar_{newTck} \cup \text{ran } lvar_{visOrig} \\
& \cup \text{ran } lvar_{hitCount} \cup \text{ran } lvar_{hits} \\
& \cup \text{ran } retvar_{BExpN}
\end{aligned}$$

**CHAN.** The channels of the HL<sup>3</sup> model are defined by the set of connected property nodes or signal nodes, respectively. Each of such a set identifies a unique variable or signal channel. Additionally, for the parameters of signals there are separate channels that carry the parameter values, such that three disjoint sets of channels  $CHN_{Var}$ ,  $CHN_{Sig}$ , and  $CHN_{SigParam}$  with  $CHN_{Var} \cap CHN_{Sig} \cap CHN_{SigParam} = \emptyset$  result:

$$\begin{aligned}
chan_{VN_{conn, max}} & : VN_{conn, max} \rightsquigarrow CHN_{Var} \\
chan_{SN_{conn, max}} & : SN_{conn, max} \rightsquigarrow CHN_{Sig} \\
vn_{conn, max, CHN_{Var}} & = chan_{VN_{conn, max}}^{-1} \\
sn_{conn, max, CHN_{Sig}} & = chan_{SN_{conn, max}}^{-1}
\end{aligned}$$

The sets of parameter channels are defined by the sets of connected signal nodes. The number of parameter channels is determined by the number of parameters of the respective signal.

$$\begin{aligned}
chan_{SN_{conn, max, param}} & : SN_{conn, max} \times \mathbb{N}_0 \rightsquigarrow CHN_{SigParam} \\
\text{dom } chan_{SN_{conn, max, param}} & = \{(snset, n) \mid \exists sn \in snset \bullet \\
& 0 \leq |paramTypes_S(sig_{SN}(sn))|\}
\end{aligned}$$

The parameter channels, like variable and signal channels, are unique, i.e.  $chan_{SN_{conn, max, param}}$  is also injective.

Then, the set of channels is the union of variable and signal channels and the signal parameter channels:

$$chan(c_{spec}) = CHN_{Var} \cup CHN_{Sig} \cup CHN_{SigParam}$$

**PORT.** The model's variable ports  $PORT_{Var}$  and signal ports  $PORT_{Sig}$  are created such that there is one port for each property node or signal node that is attached to a basic agent instance node:

$$\begin{aligned}
VN_{basic} & = \{vn \in VN_{spec} \mid \exists ain \in AIN_{basic} \bullet vn \in vn_{AIN}(tree_{AIN}(ain))\} \\
SN_{basic} & = \{sn \in SN_{spec} \mid \exists ain \in AIN_{basic} \bullet sn \in sn_{AIN}(tree_{AIN}(ain))\} \\
port_{VN_{basic}} & : VN_{basic} \rightsquigarrow PORT_{Var} \\
port_{SN_{basic}} & : SN_{basic} \rightsquigarrow PORT_{Sig}
\end{aligned}$$

$$\begin{aligned} vn_{PORT_{Var}} &= port_{VN_{basic}}^{-1} \\ sn_{PORT_{Sig}} &= port_{SN_{basic}}^{-1} \end{aligned}$$

The ports representing signal parameters  $PORT_{SigParam}$  are defined by the corresponding signal node, along with the parameter's index in the signal's parameter list:

$$\begin{aligned} port_{SN_{basic},param} &: SN_{basic} \times \mathbb{N}_0 \dashrightarrow PORT_{SigParam} \\ \text{dom } port_{SN_{basic},param} &= \{(sn, n) \mid 0 \leq |paramTypes_S(sig_{SN}(sn))|\} \end{aligned}$$

The parameter port mapping  $port_{SN_{basic},param}$  is injective, like the ones for variable and signal ports.

The set of all ports is defined as the union of the disjoint sets of variable ports, signal ports, and signal parameter ports, i.e.

$$port(c_{spec}) = Port = PORT_{Var} \cup PORT_{Sig} \cup PORT_{SigParam}$$

with  $PORT_{Var} \cap PORT_{Sig} \cap PORT_{SigParam} = \emptyset$ .

**LWP.** The set of light weight processes depends on the number of available processors within the hardware system. It cannot be defined by the transformation  $\Phi$  automatically, but has to be configured manually, such that

$$|lwp(c_{spec})| = n$$

for  $n$  available processors.

### 5.4.2 Dependencies

**Am<sub>Trans</sub>.** The mapping from transitions to abstract machines is defined by the assignment of transition nodes to agent instance nodes:

$$\begin{aligned} am_{huml,trans} &: Trans \rightarrow Am \\ t &\mapsto am_{HL3,AIN_{basic}}(ain_{MIN}(min_{TN}(tn_{trans}(t)))) \end{aligned}$$

Therefore,  $am_{trans}(c_{spec}) = am_{huml,trans}$  holds.

**Am<sub>Flow</sub>.** The assignment of flows to abstract machines for the HL<sup>3</sup> model is given by the flow expression nodes that define the flows:

$$\begin{aligned} am_{huml,flow} &: Flow \rightarrow Am \\ f &\mapsto am_{HL3,AIN_{basic}}(ain_{MIN}(min_{MEExpN}(fxn_{Flow}(f)))) \end{aligned}$$

Hence,  $am_{flow}(c_{spec}) = am_{huml,flow}$  holds.

**Subject<sub>Var</sub>.** The mapping of variables to their responsible subject is essentially given by the involved expression node:

$$\begin{aligned} subject_{ExpN_{spec}} &: ExpN_{spec} \rightarrow Flow \cup Trans \cup Am \\ expn &\mapsto \begin{cases} flow_{HL3,FlowExpN_{spec}}(expn) & ; expn \in FlowExpN_{spec} \\ trans_{HL3,TN_{spec}}(tn_{TExpN}(expn)) & ; expn \in ActExpN_{spec} \\ am_{HL3,AIN_{basic}}(ain_{ExpN}(expn)) & ; \text{else} \end{cases} \end{aligned}$$

Then, the local variables are mapped to subjects, corresponding to the expression nodes that are mapped the local variables themselves:

$$subject_{lvar} : Var \rightarrow Flow \cup Trans \cup Ifm \cup Am$$

$$lv \mapsto subject_{ExpN_{spec}}(expn)$$

with

$$\begin{aligned} & \exists (v, acc, tm) \in V \times \{read, write\} \times \{cur, prev\} \bullet \\ & \quad (expn, v, acc, tm) \mapsto lv \in lvar_{V, V_{local}} \\ & \forall \exists (s, acc) \in S \times \{read, write\} \bullet (expn, s, acc) \mapsto lv \in lvar_S \\ & \forall \exists (s, acc, n) \in S \times \{read, write\} \times \mathbb{N}_0 \bullet \\ & \quad (expn, s, acc, n) \mapsto lv \in lvar_{SigParam} \\ & \forall \exists s \in S \bullet (expn, s) \mapsto lv \in lvar_{trgIdx} \\ & \forall \exists s \in S \bullet (expn, s) \mapsto lv \in lvar_{sigIdx} \\ & \forall \exists v \in V \bullet (expn, v) \mapsto lv \in lvar_{V, \Delta t} \\ & \forall \exists v \in V \bullet (expn, v) \mapsto lv \in lvar_{V, curTck} \\ & \forall \exists v \in V \bullet (expn, v) \mapsto lv \in lvar_{V, prevTck} \\ & \forall \exists t \in tree_o HybellItem \bullet (expn, t) \mapsto lv \in lvar_{quantRes} \\ & \forall expn \mapsto lv \in lvar_{vis} \\ & \forall expn \mapsto lv \in lvar_{visSet} \\ & \forall expn \mapsto lv \in lvar_{visIdx} \\ & \forall expn \mapsto lv \in lvar_{newTck} \\ & \forall expn \mapsto lv \in lvar_{visOrig} \\ & \forall \exists t \in tree_o HybellItem \bullet (expn, t) \mapsto lv \in lvar_{hitCount} \\ & \forall \exists t \in tree_o HybellItem \bullet (expn, t) \mapsto lv \in lvar_{hits} \\ & \forall expn \mapsto lv \in retvar_{BExpN} \end{aligned}$$

The above mapping defines the subject dependency for local variables for the particular HybridUML specification:

$$subject_{var}(c_{spec}) = subject_{lvar}$$

**ChanPort.** Each port is mapped to the channel for which it provides access. This is defined by the sets of connected property or signal nodes: (1) Each node is contained in at most one maximal set of nodes, since every two sets of property nodes are united for a common property node within  $VN_{conn}$ , and (2) each node is contained at least in one such set, because every port is collected within  $VN_{conn, local}$ .

$$chan_{huml, port} : Port \rightarrow Chan$$

$$p \mapsto c$$

$$; p \in PORT_{Var} \wedge c \in CHN_{Var} \wedge vn_{PORT_{Var}}(p) \in vn_{conn, max, CHN_{Var}}(c)$$

$$p \mapsto c$$

$$; p \in PORT_{Sig} \wedge c \in CHN_{Sig} \wedge sn_{PORT_{Sig}}(p) \in sn_{conn, max, CHN_{Sig}}(c)$$

Then, this is the mapping from ports to channels:

$$chan_{port}(c_{spec}) = chan_{huml, port}$$

**InitVal<sub>port</sub>.** For the definition of initial port values, HybridUML provides two complementary concepts, that were introduced in section 3.1 – (1) “Initial Property Values” and (2) “Init State Constraints”:

*Init State Constraints.* An init state constraint is a boolean expression that defines a constraint on an agent instance node that must be satisfied in the HL<sup>3</sup> model’s initial state. All properties of the agent are available for the expression, therefore arbitrary relations on them can be postulated. This provides a powerful means to define a set of valid initial HL<sup>3</sup> channel states.

In general, such a set of constraints cannot always be solved easily, and the definition of a constraint solver (for an adequate subset of init state constraints) is out of scope of this thesis. However, we provide a mechanism to guarantee that HL<sup>3</sup> models are only executed, when the initial channel state satisfies the init state constraints. This is done by the HybridUML selector, which is defined in section 5.6.2.

*Initial Property Values.* Initial property value specifications are provided as a less powerful, but straight-forward mechanism to define the initial state of HL<sup>3</sup> channels. For each HybridUML property (of agents), an independent value specification can be given, such that mainly, a single initial channel state can be defined explicitly.

Since initial values are attached to properties, there can be more than one initial value:

$$\begin{aligned} \sigma_{CHN_{Var},initChoice} : CHN_{Var} &\rightarrow \mathcal{P}(VAL_{eval}) \\ c &\mapsto \{v \in VAL_{eval} \mid \exists vn \in c \bullet v = \sigma_{VN_{spec}}(vn)\} \end{aligned}$$

One of the provided initial values is chosen non-deterministically, if there is any:

$$\begin{aligned} \sigma_{CHN_{Var},init} : CHN_{Var} &\rightarrow VAL_{eval} \cup \{\lambda\} \\ c &\mapsto v; v \in \sigma_{CHN_{Var},initChoice}(c) \\ c &\mapsto \lambda; \sigma_{CHN_{Var},initChoice}(c) = \emptyset \end{aligned}$$

For HybridUML, every variable port of a channel initially has access to the same value – the channel’s initial value. If the channel does not provide an initial value, then a default value, corresponding to the port’s type, is used. Note that this overrides the (possible) use of init state constraints discussed above. An appropriate constraint solver would be integrated here, such that channels without initial value would get their default values then from the solver.

$$\begin{aligned} \sigma_{PORT_{Var},init} : PORT_{Var} &\rightarrow VAL_{eval} \\ p &\mapsto \sigma_{CHN_{Var},init}(chan_{huml,port}(p)) \\ &\quad ; \sigma_{CHN_{Var},init}(chan_{huml,port}(p)) \neq \lambda \\ p &\mapsto defval_{DT}(type_V(var_{VN}(vn_{PORT_{Var}}(p)))) \\ &\quad ; \sigma_{CHN_{Var},init}(chan_{huml,port}(p)) = \lambda \end{aligned}$$

The default value is deterministically defined for primitive and structured data types, and non-deterministically chosen for enumeration types:

$$\begin{aligned}
& \mathit{defval}_{DT} : DT \rightarrow VAL_{eval} \\
& \mathit{bool} \mapsto \mathit{false} \\
& \mathit{pt} \mapsto 0; \mathit{pt} \in \{\mathit{int}, \mathit{real}, \mathit{anaReal}\} \\
& \mathit{sdt} \mapsto \langle \mathit{defval}_{DT}(\mathit{varseq}_{DT}(\mathit{sdt})(1)), \dots, \\
& \quad \mathit{defval}_{DT}(\mathit{varseq}_{DT}(\mathit{sdt})(|\mathit{varseq}_{DT}(\mathit{sdt})|)) \rangle \\
& \quad ; \mathit{kind}_{DT}(\mathit{sdt}) = \mathit{struc} \\
& \mathit{et} \mapsto l; \mathit{kind}_{DT}(\mathit{et}) = DT_{enum} \wedge \mathit{dt}_L(l) = \mathit{et}
\end{aligned}$$

Signal ports initially get the value *false*, i.e. initially there is no occurrence of any signal.

$$\begin{aligned}
& \sigma_{PORT_{Sig}, \mathit{init}} : PORT_{Sig} \rightarrow VAL_{eval} \\
& p \mapsto \mathit{false}
\end{aligned}$$

The initial port data of the HL<sup>3</sup> model is then defined by the initial values above:

$$\begin{aligned}
& \mathit{data}_{PORT_{Var}, \mathit{init}} : PORT_{Var} \rightarrow Data \\
& p \mapsto \mathit{data}_{Val}(\sigma_{PORT_{Var}, \mathit{init}}(p)) \\
& \mathit{data}_{PORT_{Sig}, \mathit{init}} : PORT_{Sig} \rightarrow Data \\
& p \mapsto \mathit{data}_{Val}(\sigma_{PORT_{Sig}, \mathit{init}}(p))
\end{aligned}$$

The union of initial data for variable ports and signal ports gives the complete port initialization function:

$$\mathit{initval}_{port}(c_{spec}) = \mathit{data}_{PORT_{Var}, \mathit{init}} \cup \mathit{data}_{PORT_{Sig}, \mathit{init}}$$

**Subject<sub>port</sub>.** Every port of the model is accessible for the abstract machine *am* from which it originates, i.e. that is created from the basic agent instance node that hosts the property node or signal node, from which the port is generated. Further, all flows and transitions that are associated with *am* have access, too:

$$\begin{aligned}
& \mathit{subject}_{huml, port} : Port \rightarrow \mathcal{P}(Subject) \\
& p \mapsto \{am\} \cup \{t \in Trans \mid \mathit{am}_{huml, trans}(t) = am\} \\
& \quad \cup \{f \in Flow \mid \mathit{am}_{huml, flow}(f) = am\} \\
& \quad ; p \in PORT_{Var} \wedge \mathit{am} = \mathit{am}_{HL3, AIN_{basic}}(\mathit{ain}_{VN}(\mathit{vn}_{PORT_{Var}}(p))) \\
& \quad \quad \vee p \in PORT_{Sig} \wedge \mathit{am} = \mathit{am}_{HL3, AIN_{basic}}(\mathit{ain}_{SN}(\mathit{sn}_{PORT_{Sig}}(p)))
\end{aligned}$$

The mapping from ports to subjects is then defined as

$$\mathit{subject}_{port}(c_{spec}) = \mathit{subject}_{huml, port}$$

**SelPort.** The HybridUML selector only accesses channels (via ports) in order to deactivate signals. Therefore, it has access to the signal ports:

$$\mathit{selport}(c_{spec}) = PORT_{Sig}$$

**VisibilitySet<sub>Flow</sub>.** The publication visibility for HybridUML flows is neither restricted nor delayed. That is, for every flow, all ports are contained here such that it may publish its results to whatever recipient it may address. Further, the delay time stamp is set to zero for all entries:

$$\begin{aligned} vis_{huml,flow} &: Flow \rightarrow VisibilitySet \\ f &\mapsto \{0.0\} \times Port \end{aligned}$$

Hence,  $vis_{flow}(c_{spec}) = vis_{huml,flow}$  holds.

**VisibilitySet<sub>Ifm</sub>.** The mapping of interface modules to visibility sets is empty, since there are no interface modules:

$$vis_{ifm}(c_{spec}) = \emptyset$$

**VisibilitySet<sub>Trans</sub>.** The publication visibility for HybridUML flows is neither restricted nor delayed, similarly to flows:

$$\begin{aligned} vis_{huml,trans} &: Trans \rightarrow VisibilitySet \\ t &\mapsto \{0.0\} \times Port \end{aligned}$$

Thus,  $vis_{trans}(c_{spec}) = vis_{huml,trans}$  holds.

**Lwpsubj<sub>abs</sub>.** Since the set of light weight processes is configured manually, the assignment of abstract subjects to light weight processes  $lwp_{subj}(c_{spec})$  has to be done manually, too.

**Programs<sub>subj<sub>prog</sub></sub>.** The programs that define the behavior of the program subjects of the HL<sup>3</sup> model are created from the associated expressions. The program subjects are the flows  $f \in Flow$  and the transitions  $t \in Trans$ . There are no interface modules. The definitions for program creation from expression nodes are given in section 5.5. Here, the assignment from subjects and their respective operations to the resulting programs is defined: (1) The flows' programs are created from the expressions of the corresponding flow expression nodes, (2) the transitions' programs are based on the action expressions of the corresponding transition nodes, and of the transition expression.

$$\begin{aligned} prghuml,subj &: (Flow \times Op_{Flow}) \cup (Trans \times Op_{Trans}) \cup (Ifm \times Op_{Ifm}) \\ &\rightarrow Program \\ (f, integrate) &\mapsto prg_{ExpN,ass}(\langle fn_{Flow}(f) \rangle) \\ (t, action) &\mapsto \\ &prg_{ExpN,ass}(\text{anyseq}_{ExpN}(trg_{TN_{spec}}(tn_{trans}(t))) \frown act_{TN_{spec}}(tn_{trans}(t))) \end{aligned}$$

As a result,  $prg_{subj}(c_{spec}) = prghuml,subj$  holds.

### 5.4.3 Physical Constraints

As stated in section 1.1, HybridUML has no specification facility for the definition of physical constraints. Therefore, the system period has to be configured manually, and the period factors for flows are set to a fixed default value.

**SysPeriod.** The internal scheduling period  $\delta_{period}(c_{spec})$  has to be configured manually, and is constrained by two factors:

1. The performance of the available hardware must be sufficient, i.e. fast enough for the chosen period.
2. The chosen period must be appropriate for the simulation's requirements. Mostly, this means that it must be small enough.

Note that these constraints are conflicting in general. Therefore, it is not ensured that a valid period  $\delta_{period}(c_{spec})$  can be found.

**PeriodFlow.** By default, every flow is scheduled with the system period  $\delta_{period}(c_{spec})$ , i.e. the factor is set to 1 for all flows:

$$\begin{aligned} period_{huml,flow} : Flow &\rightarrow \mathbb{N} \\ f &\mapsto 1 \end{aligned}$$

This mapping can be adjusted manually. Anyway,  $period_{flow}(c_{spec}) = period_{huml,flow}$  holds.

**PeriodIfm,poll.** Since there are no interface modules, the mapping from interface modules to period factors for the operation *poll* is empty:

$$period_{ifm,poll}(c_{spec}) = \emptyset$$

**PeriodIfm,tmit.** Similarly, the mapping from interface modules to period factors for the operation *transmit* is empty:

$$period_{ifm,tmit}(c_{spec}) = \emptyset$$

## 5.5 Code Creation for Expression Nodes

This section defines the creation of program code for sequences of expression nodes. The resulting code defines the execution semantics of a sequence of expressions, within their expression node context. In particular, the semantics of single expression nodes are therefore given.

Exemplarily, for selected generation rules, corresponding program code is pointed out in appendix D.2. There, from the implementation of the transformation  $\Phi_{HUML}$ , C++ operations are given, which correspond to programs of the HL<sup>3</sup> model of the case study “Radio-Based Train Control”.

Note that the C++ code differs from the formally defined programs  $p \in Program$  in details. For example, declaration of local variables is done within the code, not separately, and initialization is merged with declaration. Further, some expressions and statements are represented more conveniently, like `for`-statements, instead of `while`.

**Program Strings.** The transformation rules given in this section define code by giving *program strings*, that are sequences  $s \in String$  of characters. For convenience, we use the conventional string notation "abc" from programming languages as a shorthand notation for the sequence  $\langle a, b, c \rangle$ .

For the representation of literals, local variables, and ports we assume an injective mapping

$$string : VAL_{eval} \cup Var \cup Port \hookrightarrow String$$

that defines a unique string representation for each literal, variable, and port.

In terms of the HL<sup>3</sup> operational semantics, programs are represented as sequences  $p \in Program$  of *program statements*. The mapping

$$prgString : String \rightarrow Program$$

from program strings to sequences of statements is straight-forward, using the character ";" as statement delimiter, as well as the standard block notation for conditional and loop statements, ending with a closing brace "}". We omit the formal definition here.

**Evaluation and Assignment Interpretation of Expressions.** There are two different interpretations of expressions in the context of HybridUML specifications:

*Evaluation Interpretation* Some expressions have an evaluation interpretation, such that the corresponding program calculates a result value when it is executed. The result value of complete expressions that are contained in expression nodes is always of boolean type, whereas contained sub-expressions of course can have other types, as defined in chapter 3.

(1) Trigger expressions, (2) guard expressions, and (3) invariant constraint expressions have evaluation interpretations that determine whether a trigger is active, or a guard or an invariant constraint is satisfied, respectively.

*Assignment Interpretation* The assignment interpretation of expressions defines programs that are executed like procedure calls and do not return a result value. Those programs will typically calculate new values for HybridUML variables or set/unset HybridUML signals and publish the new values on the respective channels.

The affected kinds of expressions are (1) action expressions, (2) flow (and algebraic) constraint expressions, and (3) trigger expressions. The effect of the assignment interpretation of triggers is that the corresponding signal is unset after it has been consumed.

The definition of the resulting programs given below is structured accordingly: On the basis of the hybrid item trees and identifier item trees which represent (sub-)expressions, the mappings below are defined mutually recursively, such that particularly assignment code can contain evaluation code for the calculation of new values.

**Evaluation Code of Expression Nodes.** For a sequence of expression nodes, the code that implements the evaluation interpretation is given by the function

$$prgExpN_{eval} : seq ExpN_{spec} \rightarrow Program$$

For expression nodes with contained expression of boolean type, i.e. for

$$\forall \text{expn} \in \{\text{expn}_1, \dots, \text{expn}_n\} \bullet \exists((t, r), \text{val}), \text{sub}) \in \text{tree}_o \text{HybelItem} \bullet \\ \text{ht}_{\text{expn}}(\text{expn}) = (((t, r), \text{val}), \text{sub}) \wedge t = \text{bool} \wedge r \in \{\text{op}, \forall, \exists, \text{var}, \text{lit}\}$$

the function  $\text{prg}_{\text{ExpN}, \text{eval}}$  is defined as

$$\langle \text{expn}_1, \dots, \text{expn}_n \rangle \mapsto \text{prgString}(\text{initcode}_{V, \text{cur}}(\{\text{expn}_1, \dots, \text{expn}_n\}) \hat{\ } \\ \text{initcode}_{V, \text{prev}}(\{\text{expn}_1, \dots, \text{expn}_n\}) \hat{\ } \text{initcode}_S(\{\text{expn}_1, \dots, \text{expn}_n\}) \hat{\ } \\ \text{pre}(\text{ht}_{\text{expn}}(\text{expn}_1), \text{cur}, \text{expn}_1) \hat{\ } \dots \hat{\ } \text{pre}(\text{ht}_{\text{expn}}(\text{expn}_n), \text{cur}, \text{expn}_n) \\ \hat{\ } \text{string}(\text{retvar}_{\text{BExpN}}(\text{expn}_1)) \hat{\ } \text{":="} \hat{\ } \\ \text{main}(\text{ht}_{\text{expn}}(\text{expn}_1), \text{cur}, \text{expn}_1) \\ \hat{\ } \text{"} \wedge \text{"} \hat{\ } \dots \hat{\ } \text{"} \wedge \text{"} \hat{\ } \\ \text{main}(\text{ht}_{\text{expn}}(\text{expn}_n), \text{cur}, \text{expn}_n) \\ \hat{\ } \text{"}, \text{"} \hat{\ } \\ \text{post}(\text{ht}_{\text{expn}}(\text{expn}_1), \text{cur}, \text{expn}_1) \hat{\ } \dots \hat{\ } \text{post}(\text{ht}_{\text{expn}}(\text{expn}_n), \text{cur}, \text{expn}_n))$$

At the beginning, local variables are initialized which are accessed by the expressions of the expression nodes. For the evaluation code, (1) current values of HybridUML variables, (2) previous values of HybridUML variables, and (3) signal states, i.e. enabledness of signals are read from the respective channels and stored locally in dedicated HL<sup>3</sup> variables. The actual evaluation of an expression consists of up to three parts:

*Main Code* The main evaluation code is an expression in terms of while-programs. Simple HybEL expressions are represented directly by such an expression; for example, the boolean literal *true* is represented as the program expression "true".

In contrast, more complex HybEL expressions are implemented by a sequential sub-program and cannot be given by a simple program expression. Examples are the quantifiers  $\forall, \exists$  – they are mapped to while-loops which store the result in a local variable *result*. Here, "result" constitutes the main code of the quantified HybEL expression, whereas the sub-program that is run before is the *pre-code*.

*Pre-Code* The evaluation pre-code subsumes all sub-programs (concatenated sequentially) that are needed in order to prepare the expression of the main code.

*Post-Code* For expressions that require some functionality to be effective *after* the evaluation is done, the post-code can be created and filled with the respective program string. This will be used for the *assignment interpretation* of expressions, and exists here for the evaluation interpretation only for consistency.

Therefore, succeeding the initialization code, the rest of the program consists of the three code parts described above. First, pre-code is inserted per expression node. Then, the main code is inserted for all expression nodes, combined by logical conjunction; and the resulting value is assigned to the dedicated return variable of the first expression node. Finally, the post-code is appended for all expression nodes.

For expression nodes that contain an expression which is not of boolean type, the function  $prg_{ExpN,eval}$  provides the empty program:

$$\langle expn_1, \dots, expn_n \rangle \mapsto \langle \rangle$$

**Assignment Code of Expression Nodes.** The assignment code that is created for a sequence of expression nodes is defined as function

$$prg_{ExpN,ass} : seq\ ExpN_{spec} \rightarrow Program$$

Expressions to be interpreted for assignment must be HybEL assignment expressions, therefore

$$\begin{aligned} \forall expn \in \{expn_1, \dots, expn_n\} \bullet \exists((t, r), val), sub) \in tree_o\ HybellItem \bullet \\ ht_{expn}(expn) = (((t, r), val), sub) \wedge r \in \{ass, diffAss, intNondetAss, \\ assGroup, diffAssGroup, sendSig\} \end{aligned}$$

is required. Then, the function  $prg_{ExpN,ass}$  is defined as

$$\begin{aligned} \langle expn_1, \dots, expn_n \rangle \mapsto prgString( \\ \quad \textit{initcode}_{visparam}(\{expn_1, \dots, expn_n\}) \hat{\ } \\ \quad \textit{initcode}_{V,cur}(\{expn_1, \dots, expn_n\}) \hat{\ } \\ \quad \textit{initcode}_{V,prev}(\{expn_1, \dots, expn_n\}) \hat{\ } \\ \quad \textit{initcode}_{V,curTck}(\{expn_1, \dots, expn_n\}) \hat{\ } \\ \quad \textit{initcode}_{V,prevTck}(\{expn_1, \dots, expn_n\}) \hat{\ } \\ \quad \textit{initcode}_S(\{expn_1, \dots, expn_n\}) \hat{\ } \textit{initcode}_{SigParam}(\{expn_1, \dots, expn_n\}) \hat{\ } \\ \quad \textit{asspre}(ht_{expn}(expn_1), expn_1) \hat{\ } \dots \hat{\ } \textit{asspre}(ht_{expn}(expn_n), expn_n) \hat{\ } \\ \quad \textit{assmain}(ht_{expn}(expn_1), expn_1) \hat{\ } \dots \hat{\ } \textit{assmain}(ht_{expn}(expn_n), expn_n) \hat{\ } \\ \quad \textit{pubcode}_V(\{expn_1, \dots, expn_n\}) \hat{\ } \\ \quad \textit{pubcode}_{S,send}(\{expn_1, \dots, expn_n\}) \hat{\ } \\ \quad \textit{pubcode}_{SigParam}(\{expn_1, \dots, expn_n\}) \hat{\ } \\ \quad \textit{pubcode}_{S,recv}(\{expn_1, \dots, expn_n\}) \hat{\ } \\ \quad \textit{asspost}(ht_{expn}(expn_1), expn_1) \hat{\ } \dots \hat{\ } \textit{asspost}(ht_{expn}(expn_n), expn_n)) \end{aligned}$$

The creation of assignment code is somewhat similar to the creation of evaluation code. The generated code starts with the initialization of the visibility set variable from the visibility set parameter, followed by the initialization of local variables from the respective channels. In addition to the current and previous values of HybridUML variables and HybridUML signal states, the time stamps of the current and previous values of HybridUML variables, and the parameter values of HybridUML signals are read and stored locally.

It is followed by the pre-code, which is appended by the main code. The main *assignment* code of an expression node is a program statement, rather than a program expression; therefore, there is no conjunction and assignment, but just a sequence of statements.

Before the post-code is appended, the *publication code* is added: Since the aim of assignment code is to calculate *and publish* new values for channels, this code publishes the calculated values from local variables to the respective channels. Note that the post-code therefore is effective *after* the publication of results.

For expression nodes that contain an expression which is not an assignment, the function  $prg_{ExpN, ass}$  provides the empty program:

$$\langle expn_1, \dots, expn_n \rangle \mapsto \langle \rangle$$

### 5.5.1 Evaluation Interpretation of Hybel Item Trees

In the context of an expression node, a hybel item tree defines program strings that implement the evaluation of the encoded expression. This is defined by the mapping

$$code : tree_o HybelItem \times \{cur, prev\} \times ExpN \rightarrow String^3$$

The distinction between  $cur$  and  $prev$  is made for the contained variables, for which normally the currently available value is effective for calculations, but in specific contexts the previous one. The main code, pre-code and post-code parts are given by the mappings

$$main : tree_o HybelItem \times \{cur, prev\} \times ExpN \rightarrow String$$

$$twn \mapsto \pi_1 code(twn)$$

$$pre : tree_o HybelItem \times \{cur, prev\} \times ExpN \rightarrow String$$

$$twn \mapsto \pi_2 code(twn)$$

$$post : tree_o HybelItem \times \{cur, prev\} \times ExpN \rightarrow String$$

$$twn \mapsto \pi_3 code(twn)$$

The definition of  $code$  is then given recursively, by the structure of the hybel item trees.

**Literals.** Literals of all types are mapped to their string representations:

$$(((t, lit), val), sub), tm, expn) \mapsto (string(val), "", "")$$

$$; t \in \{bool, int, real\} \cup DT_{enum}$$

$$(((sdt_{anon}, lit), val), \langle s_1, \dots, s_n \rangle), tm, expn) \mapsto$$

$$(" \{ " \wedge main(s_1, tm, expn) \wedge " \wedge \dots \wedge " \wedge main(s_n, tm, expn) \wedge " \} ",$$

$$pre(s_1, tm, expn) \wedge \dots \wedge pre(s_n, tm, expn),$$

$$post(s_1, tm, expn) \wedge \dots \wedge post(s_n, tm, expn))$$

*Examples:* Almost all examples of section D.2 contain literals; e.g. D.2.9, D.2.11.

**Variables and Signals.** The string representation of variables, as well as of received signals (i.e. triggers) is solely given by the contained identifier item tree. For sent signals, there is no interpretation within read-only context.

$$(((t, var), idtree), sub), tm, expn) \mapsto code_{Id}(idtree, tm, expn)$$

$$; t \in DT \cup \{recvSig\}$$

*Examples:* D.2.12, D.2.13, D.2.14, D.2.17, D.2.18, D.2.19.

**Derivative Values of Variables.** Reading of a derivative is implemented by calculating the last integration step, from the current and previous value and their time stamps of the corresponding analog real variable.

```

(((anaReal, derivVar), idtree), sub), tm, expn)  $\mapsto$ 

(("  $\wedge$  mainId(idtree, cur, expn)  $\wedge$  " - "  $\wedge$  mainId(idtree, prev, expn)
 $\wedge$  ") / "  $\wedge$  string(lvarV,Δt(expn, v))  $\wedge$  " ["  $\wedge$  idxmain  $\wedge$  "]") ,

idxpre  $\wedge$ 
string(lvarV,Δt(expn, v))  $\wedge$  "[0] := "  $\wedge$ 
"left("  $\wedge$  string(lvarV,curTck(expn, v))  $\wedge$  "[0]"  $\wedge$ 
" - left("  $\wedge$  string(lvarV,prevTck(expn, v))  $\wedge$  "[0]"; "
 $\wedge$  ...  $\wedge$  repeat for 1..n
string(lvarV,Δt(expn, v))  $\wedge$  " ["  $\wedge$  string(n)  $\wedge$  "] := "  $\wedge$ 
"left("  $\wedge$  string(lvarV,curTck(expn, v))  $\wedge$  " ["  $\wedge$  string(n)  $\wedge$  "]"  $\wedge$ 
" - left("  $\wedge$  string(lvarV,prevTck(expn, v))  $\wedge$  " ["  $\wedge$  string(n)  $\wedge$  "]" ); "
 $\wedge$ 
"if("  $\wedge$  string(lvarV,Δt(expn, v))  $\wedge$  "[0]  $\leq$  0)"  $\wedge$ 
"{return; }"
 $\wedge$  ...  $\wedge$  repeat for 1..n
"if("  $\wedge$  string(lvarV,Δt(expn, v))  $\wedge$  " ["  $\wedge$  string(n)  $\wedge$  "]  $\leq$  0)"  $\wedge$ 
"{return; }" ,

idxpost)

with
idtree = ((tid, (v, acc)), subid)  $\wedge$  n = |vnexpn(v, expn)| - 1  $\wedge$ 
(subid =  $\langle\langle$ (indexExp, exptree),  $\langle\rangle$  $\rangle\rangle$ 
 $\Rightarrow$  (idxmain, idxpre, idxpost) = code(exptree, tm, expn))
 $\wedge$  (subid =  $\langle\rangle$ )  $\Rightarrow$  (idxmain, idxpre, idxpost) = ("0", "", "")

```

The pre-code prepares the time difference for each of the variable's indices; this is necessary, because the variable's index expression dynamically decides which index is accessed during execution. Further, a check is included to ensure that the time difference is positive, otherwise the calculation will be skipped. Therefore, a division by zero is avoided. The main code just calculates the integration step.

**Unary Operations.** The code for the negation operation, which is the single available unary operation, is based on the code for the contained expression. Additionally, the negation operator is applied:

```

(((bool, op),  $\neg$ ),  $\langle s \rangle$ ), tm, expn)  $\mapsto$ 

("  $\neg$ "  $\wedge$  main(s, tm, expn), pre(s, tm, expn), post(s, tm, expn))

```

*Examples:* D.2.14, D.2.12, D.2.19

**Binary Operations.** Most binary operations directly rely on the contained expressions, i.e. the left-hand side and the right-hand side are transformed to programs separately first, and then combined with the corresponding operator within the program, in a similar fashion as for unary operations:

$$\begin{aligned}
& (((t, r), \diamond), \langle (((t_1, r_1), val_1), sub_1), (((t_2, r_2), val_2), sub_2)) \rangle, tm, expn) \mapsto \\
& \text{"("} \wedge \text{main}(\langle (((t_1, r_1), val_1), sub_1) \wedge \text{"} \diamond \text{"} \wedge \text{main}(\langle (((t_2, r_2), val_2), sub_2) \wedge \text{"} \rangle), \\
& \text{pre}(\langle (((t_1, r_1), val_1), sub_1) \wedge \text{pre}(\langle (((t_2, r_2), val_2), sub_2) \wedge \text{post}(\langle (((t_1, r_1), val_1), sub_1) \wedge \text{post}(\langle (((t_2, r_2), val_2), sub_2) \wedge \\
& \text{with}\{t_1, t_2\} \cap \{\text{real}, \text{anaReal}\} = \emptyset \vee \diamond \notin \{=, \neq\}
\end{aligned}$$

*Examples:* D.2.12, D.2.13, D.2.14, D.2.17, D.2.18, D.2.19.

**Equality Comparison for Real Values.** There are two binary operations that need special treatment – equality comparison and inequality comparison of real values is problematic. Since the execution of the HL<sup>3</sup> model is discretized, it is not guaranteed that *zero crossings* of values  $x_1, x_2$  are detected, i.e. that during the discretized run,  $x_1 = x_2$  is observed, when their continuous evolutions would cross. Depending on the chosen system period  $\delta_{period}(c_{spec})$  and the (implemented) precision of the representation of real values, it is rather unlikely that a naively-implemented (in-)equality comparison would succeed.

Therefore, as an approximation, we compare not only the current values, but also the previous ones, such that a zero crossing is detected, whenever between two succeeding discrete evaluation steps, *no second* zero crossing occurs. Therefore, we actually check the condition

$$x_{1,prev} \leq x_{2,cur} \wedge x_{1,cur} \geq x_{2,prev} \vee x_{1,prev} \geq x_{2,cur} \wedge x_{1,cur} \leq x_{2,prev}$$

Thus, hybel item trees representing (in-)equality comparisons are mapped to programs implementing this check; they are defined by

$$\begin{aligned}
& (((t, r), ==), \langle (((t_1, r_1), val_1), sub_1), (((t_2, r_2), val_2), sub_2)) \rangle, tm, expn) \mapsto \\
& \text{"(((("} \wedge \\
& \text{main}(\langle (((t_1, r_1), val_1), sub_1), prev, expn) \wedge \text{"} \geq \text{"} \wedge \\
& \text{main}(\langle (((t_2, r_2), val_2), sub_2), cur, expn) \wedge \text{"} \wedge \text{"} \wedge \\
& \text{main}(\langle (((t_1, r_1), val_1), sub_1), cur, expn) \wedge \text{"} \leq \text{"} \wedge \\
& \text{main}(\langle (((t_2, r_2), val_2), sub_2), prev, expn) \wedge \text{"} \vee \text{"} \wedge \\
& \text{main}(\langle (((t_1, r_1), val_1), sub_1), prev, expn) \wedge \text{"} \leq \text{"} \wedge \\
& \text{main}(\langle (((t_2, r_2), val_2), sub_2), cur, expn) \wedge \text{"} \wedge \text{"} \wedge \\
& \text{main}(\langle (((t_1, r_1), val_1), sub_1), cur, expn) \wedge \text{"} \geq \text{"} \wedge \\
& \text{main}(\langle (((t_2, r_2), val_2), sub_2), prev, expn) \wedge \\
& \text{"))))", \\
& \text{"", ""}
\end{aligned}$$

with  $\{t_1, t_2\} \cap \{real, anaReal\} \neq \emptyset$

Inequality comparison programs are based on equality comparison:

$$\begin{aligned} &(((t, r), \neq), \langle \langle \langle (t_1, r_1), val_1 \rangle, sub_1 \rangle, \langle \langle (t_2, r_2), val_2 \rangle, sub_2 \rangle \rangle \rangle, tm, expn) \mapsto \\ &(" \neq " \wedge main(\langle \langle \langle (t, r), == \rangle, \langle \langle (t_1, r_1), val_1 \rangle, sub_1 \rangle, \langle \langle (t_2, r_2), val_2 \rangle, sub_2 \rangle \rangle \rangle), \\ &tm, expn), \\ &pre(\langle \langle \langle (t, r), == \rangle, \langle \langle (t_1, r_1), val_1 \rangle, sub_1 \rangle, \langle \langle (t_2, r_2), val_2 \rangle, sub_2 \rangle \rangle \rangle, tm, expn), \\ &post(\langle \langle \langle (t, r), == \rangle, \langle \langle (t_1, r_1), val_1 \rangle, sub_1 \rangle, \langle \langle (t_2, r_2), val_2 \rangle, sub_2 \rangle \rangle \rangle, tm, expn)) \end{aligned}$$

with  $\{t_1, t_2\} \cap \{real, anaReal\} \neq \emptyset$

**Quantified Boolean Expressions.** Quantified boolean expressions are implemented by **while** loops over the specified integer ranges. For each integer range, a separate loop is created. Within the loop, the bound expression is evaluated, and the boolean result is adapted and stored in a dedicated variable. This is encoded in the pre-code part; the main code then only contains the result variable:

$$\begin{aligned} &(quantTree, tm, expn) \mapsto \\ &(string(lvar_{quantRes}(expn, quantTree)), \\ &asspre_{Id}(idtree_{bnd}, tm, expn) \wedge pre(s_{low,1}, tm, expn) \\ &\wedge pre_{Id}(idtree_{bnd}, tm, expn) \wedge pre(s_{up,1}, tm, expn) \\ &\wedge pre(boundExp, tm, expn) \wedge asspre_{Id}(idtree_{bnd}, tm, expn) \\ &\wedge \dots \wedge \text{repeat for } 1..n \\ &asspre_{Id}(idtree_{bnd}, tm, expn) \wedge pre(s_{low,n}, tm, expn) \\ &\wedge pre_{Id}(idtree_{bnd}, tm, expn) \wedge pre(s_{up,n}, tm, expn) \\ &\wedge pre(boundExp, tm, expn) \wedge asspre_{Id}(idtree_{bnd}, tm, expn) \wedge \\ &string(lvar_{quantRes}(expn, quantTree)) \wedge " := " \wedge initial \wedge "; " \wedge \\ &assmain_{Id}(idtree_{bnd}, tm, expn) \wedge " := " \wedge main(s_{low,1}, tm, expn) \wedge "; " \wedge \\ &" \text{while} (" \wedge main_{Id}(idtree_{bnd}, tm, expn) \wedge " \leq " \wedge main(s_{up,1}, tm, expn) \\ &\wedge ") \wedge " \wedge sign \wedge string(lvar_{quantRes}(expn, quantTree)) \wedge ") \{ " \wedge \\ &string(lvar_{quantRes}(expn, quantTree)) \wedge " := " \wedge main(boundExp, tm, expn) \\ &\wedge "; " \wedge " ++ " \wedge assmain_{Id}(idtree_{bnd}, tm, expn) \wedge "; " \\ &\wedge \dots \wedge \text{repeat for } 1..n \\ &assmain_{Id}(idtree_{bnd}, tm, expn) \wedge " := " \wedge main(s_{low,n}, tm, expn) \wedge "; " \wedge \\ &" \text{while} (" \wedge main_{Id}(idtree_{bnd}, tm, expn) \wedge " \leq " \wedge main(s_{up,n}, tm, expn) \\ &\wedge ") \wedge " \wedge sign \wedge string(lvar_{quantRes}(expn, quantTree)) \wedge ") \{ " \wedge \\ &string(lvar_{quantRes}(expn, quantTree)) \wedge " := " \wedge main(boundExp, tm, expn) \\ &\wedge "; " \wedge " ++ " \wedge assmain_{Id}(idtree_{bnd}, tm, expn) \wedge "; " , \\ &asspost_{Id}(idtree_{bnd}, tm, expn) \wedge post(s_{low,1}, tm, expn) \end{aligned}$$

$$\begin{aligned}
& \hat{\sim} post_{Id}(idtree_{bnd}, tm, expn) \hat{\sim} post(s_{up,1}, tm, expn) \\
& \hat{\sim} post(boundExp, tm, expn) \hat{\sim} asspost_{Id}(idtree_{bnd}, tm, expn) \\
& \hat{\sim} \dots \hat{\sim} \text{repeat for } 1..n \\
& asspost_{Id}(idtree_{bnd}, tm, expn) \hat{\sim} post(s_{low,n}, tm, expn) \\
& \hat{\sim} post_{Id}(idtree_{bnd}, tm, expn) \hat{\sim} post(s_{up,n}, tm, expn) \\
& \hat{\sim} post(boundExp, tm, expn) \hat{\sim} asspost_{Id}(idtree_{bnd}, tm, expn)
\end{aligned}$$

with

$$\begin{aligned}
quantTree = & (((bool, q), idtree_{bnd}), (((intSet, intSpecs), val), \\
& \langle \langle \langle (intSet, intRange), val_1 \rangle, \langle s_{low,1}, s_{up,1} \rangle \rangle, \dots, \\
& \langle \langle (intSet, intRange), val_n \rangle, \langle s_{low,n}, s_{up,n} \rangle \rangle \rangle), boundExp))
\end{aligned}$$

$$\wedge q \in \{\forall, \exists\}$$

$$\wedge q = \forall \Rightarrow \text{initval} = \text{"true"} \wedge \text{sign} = \text{"+"}$$

$$\wedge q = \exists \Rightarrow \text{initval} = \text{"false"} \wedge \text{sign} = \text{"-"}$$

*Examples:* D.2.12, D.2.13, D.2.14, D.2.17, D.2.18, D.2.19.

**Assignment Expressions.** Assignment expressions that assign values to variables apply the *assignment interpretation* for the left-hand side, which represents the variable to be assigned. The right-hand side is an expression that is interpreted as a “conventional” expression, i.e. the read interpretation is used, recursively. This rule excludes differential assignments that assign a derivative:

$$(((t, r), val), \langle s_1, \dots, s_n \rangle), tm, expn) \mapsto$$

$$(assmain(s_1, expn) \hat{\sim} \text{" := " } \hat{\sim} main(s_2, tm, expn),$$

$$asspre(s_1, expn) \hat{\sim} pre(s_2, tm, expn),$$

$$asspost(s_1, expn) \hat{\sim} post(s_2, tm, expn))$$

$$\text{with } s_1 = ((role_1, val_1), sub_1) \wedge role_1 \neq (anaReal, derivVar)$$

$$\wedge r \in \{ass, diffAss\}$$

*Examples:* D.2.3, D.2.4, D.2.9, D.2.11.

**Assignment of Derivative Values of Variables.** The assignment of the derivative of an analog real variable  $v$  of the form  $\dot{v} = val$  is interpreted such that for the next integration step, the calculated value from the right-hand side is the constant slope for this integration step. That is, for the approximation of the derivative  $\dot{v}$

$$\begin{aligned}
\dot{v}(t) &= \lim_{\Delta t \rightarrow 0} \frac{v(t + \Delta t) - v(t)}{\Delta t} \\
&\approx \frac{v(t + \Delta t) - v(t)}{\Delta t} \\
&= \frac{v(t_{new}) - v(t)}{t_{new} - t}
\end{aligned}$$

a new value for  $v$  results:

$$\begin{aligned} v(t_{new}) &= val \cdot (t_{new} - t) + v(t) \\ &\approx \dot{v}(t) \cdot (t_{new} - t) + v(t) \end{aligned}$$

Since there are possibly different publication times  $t_{new}$ , given by the visibility set parameter of the flow that hosts the assignment, this calculation has to be done for each visibility entry within that set. Therefore, a **while** loop is generated that iterates the visibility entries and repeats the calculation of the integration step, correspondingly.

```

((((anaReal, diffAss), val),
  (((anaReal, derivVar), idtree1), sub1), s2, . . . , sn)), tm, expn) ↦

(assmain((((anaReal, derivVar), idtree1), sub1), expn) ^ ":= ("
  ^ main(s2, tm, expn) ^ ") \cdot (left(" ^ string(lvarnewTck(expn))
  ^ ") - left(" ^ string(lvarV,curTck(expn, v)) ^ ") + "
  ^ main((((anaReal, derivVar), idtree1), sub1), tm, expn),

pre((((anaReal, derivVar), idtree1), sub1), tm, expn)
  ^ asspre((((anaReal, derivVar), idtree1), sub1), expn)
  ^ pre(s2, tm, expn) ^

string(lvarvisOrig(expn)) ^ ":= getVisParam();" ^
string(lvarvisIdx(expn)) ^ ":= size(" ^ string(lvarvisSet(expn)) ^ ");" ^
  ^ "while(" ^ string(lvarvisIdx(expn)) ^ "> 0){" ^
"clear(" ^ string(lvarvisSet(expn)) ^ ");" ^
string(lvarvis(expn)) ^ ":= getEntry(" ^ string(lvarvisOrig(expn))
  ^ "," ^ string(lvarvisIdx(expn)) ^ ");" ^
"addEntry(" ^ string(lvarvisSet(expn))
  ^ "," ^ string(lvarvis(expn)) ^ ");" ^
string(lvarnewTck(expn)) ^ ":= left(" ^ string(lvarvis(expn)) ^ ");" ^

post((((anaReal, derivVar), idtree1), sub1), tm, expn)
  ^ asspost((((anaReal, derivVar), idtree1), sub1), expn)
  ^ post(s2, tm, expn)
  ^ "--" ^ string(lvarvisIdx(expn)) ^ "; }")

with idtree1 = ((t, (v, acc)), subid)

```

Note that the main code is the calculation of one single integration step, whereas the pre-code and post-code model the surrounding **while** loop: The pre-code prepares the iteration of the visibility entries given by the visibility set parameter and defines the condition of the loop. The post-code increments the counter for the visibility entries and closes the loop. Here it is assumed that between main code and post-code some send code is inserted that actually publishes the calculated value for the current iteration; this is defined above, by  $prgExpN, ass$ .

*Examples:* D.2.1.

**Default Rule.** For all other hybel item trees that are not mapped above, an empty program is created:

$$(tree, tm, expn) \mapsto ("", "", ""); \text{ else}$$

## 5.5.2 Evaluation Interpretation of Identifier Item Trees

For identifiers (including index expressions), the evaluation interpretation maps identifier item trees to program strings. Similarly to hybel item trees, this also depends on the expression node for which the program is created, and on the flag that either requests the current or the previous value for the identifier.

$$code_{Id} : tree_o IdItem \times \{cur, prev\} \times ExpN \rightarrow String^3$$

There are main code, pre-code, and post-code projections, given by the mappings

$$\begin{aligned} main_{Id} &: tree_o IdItem \times \{cur, prev\} \times ExpN \rightarrow String \\ & \quad tvn \mapsto \pi_1 code_{Id}(tvn) \\ pre_{Id} &: tree_o IdItem \times \{cur, prev\} \times ExpN \rightarrow String \\ & \quad tvn \mapsto \pi_2 code_{Id}(tvn) \\ post_{Id} &: tree_o IdItem \times \{cur, prev\} \times ExpN \rightarrow String \\ & \quad tvn \mapsto \pi_3 code_{Id}(tvn) \end{aligned}$$

**Variable Identifiers.** A variable identifier with index expression is mapped to the local variable that represents *read* access to the HybridUML variable or signal. For every identifier, the local variable is of array type, and the array index results from the index expression:

$$\begin{aligned} &(((t, (v, acc)), \langle (indexExp, exptree), \langle \rangle \rangle), tm, expn) \mapsto \\ & (string(lvar_{V, V_{local}}(expn, v, read, tm)) \\ & \quad \wedge "[\wedge main_{Id}(\langle (indexExp, exptree), \langle \rangle \rangle, tm, expn) \wedge ]"), \\ & pre_{Id}(\langle (indexExp, exptree), \langle \rangle \rangle, tm, expn), \\ & post_{Id}(\langle (indexExp, exptree), \langle \rangle \rangle, tm, expn) \\ & \text{with } t \in DT \end{aligned}$$

Variable identifiers without index expression are treated similarly, but array index 0 is used. Therefore, variable expressions  $x$  and  $x[0]$  have the same meaning.

$$\begin{aligned} &(((t, (v, acc)), \langle \rangle), tm, expn) \mapsto \\ & (string(lvar_{V, V_{local}}(expn, v, read, tm)) \wedge "[0]", "", "") \\ & \text{with } t \in DT \end{aligned}$$

*Examples:* D.2.12, D.2.13, D.2.14, D.2.17, D.2.18, D.2.19.

**Sub-Variables of Structured Data Types.** Access to a sub-variable of a variable of structured data type is defined recursively:

$$\begin{aligned}
&(((t, (v, acc)), \langle s_1, s_2 \rangle), tm, expn) \mapsto \\
&(string(lvar_{V, V_{local}}(expn, v, read, tm)) \hat{\ } " [ " \hat{\ } main_{Id}(s_1, tm, expn) \hat{\ } " ] . " \\
&\hat{\ } main_{Id}(s_2, tm, expn), \\
&pre_{Id}(s_1, tm, expn) \hat{\ } pre_{Id}(s_2, tm, expn), \\
&post_{Id}(s_1, tm, expn) \hat{\ } post_{Id}(s_2, tm, expn)) \\
&\text{with } t \in DT_{struct}
\end{aligned}$$

If there is no index expression, index 0 is applied:

$$\begin{aligned}
&(((t, (v, acc)), \langle (t_s, val_s), sub_s \rangle), tm, expn) \mapsto \\
&(string(lvar_{V, V_{local}}(expn, v, read, tm)) \hat{\ } "[0] . " \\
&\hat{\ } main_{Id}(\langle (t_s, val_s), sub_s \rangle, tm, expn), \\
&pre_{Id}(\langle (t_s, val_s), sub_s \rangle, tm, expn), \\
&post_{Id}(\langle (t_s, val_s), sub_s \rangle, tm, expn)) \\
&\text{with } t \in DT_{struct} \wedge t_s \neq indexExp
\end{aligned}$$

*Examples:* D.2.12, D.2.13, D.2.14, D.2.17, D.2.18, D.2.19.

**Index Expressions of Triggers.** Triggers can have index assignment subexpressions, which require special treatment. In contrast, triggers with conventional index expressions are mapped in the same fashion as variables are:

$$\begin{aligned}
&(((recvSig, (s, recv)), \langle (indexExp, (\langle (int, r), val), sub), \langle \rangle \rangle), tm, expn) \mapsto \\
&(string(lvar_S(expn, s, read)) \\
&\hat{\ } " [ " \hat{\ } main_{Id}(\langle (indexExp, (\langle (int, r), val), sub), \langle \rangle \rangle, tm, expn) \hat{\ } " ] " , \\
&pre_{Id}(\langle (indexExp, (\langle (int, r), val), sub), \langle \rangle \rangle, tm, expn), \\
&post_{Id}(\langle (indexExp, (\langle (int, r), val), sub), \langle \rangle \rangle, tm, expn)) \\
&\text{with } r \neq indexAss
\end{aligned}$$

If there is no index expression, index 0 is applied:

$$\begin{aligned}
&(((recvSig, (s, recv)), \langle \rangle), tm, expn) \mapsto \\
&(string(lvar_S(expn, s, read)) \hat{\ } "[0]" , " " , " ")
\end{aligned}$$

For triggers with index assignment subexpression, a program is created that accepts signal occurrences *for all* indices. The succeeding evaluation of the actually affected index is defined by the *assignment interpretation* of triggers, which is introduced later.

$$\begin{aligned}
&(((recvSig, (s, recv)), \\
&\langle (indexExp, (\langle (int, indexAss), val), sub), \langle \rangle \rangle), tm, expn) \mapsto \\
&(" ( " \hat{\ } string(lvar_S(expn, s, read)) \hat{\ } "[0]" ,
\end{aligned}$$

$\hat{\ } \vee \hat{\ } \wedge \dots \hat{\ } \vee \hat{\ } \wedge$  repeat for  $1..n$   
 $string(lvar_S(expn, s, read)) \hat{\ } \wedge [string(n)]$ ,  
 $"" , ""$   
 with  $n = |sn_{expn}(s, expn)| - 1$

**Conventional Index Expressions.** Conventional index expressions of identifier items are given by hybel item trees:

$$(((indexExp, exptree), sub), tm, expn) \mapsto code(exptree, tm, expn)$$

*Examples:* D.2.12, D.2.13, D.2.14, D.2.17, D.2.18, D.2.19.

**Default Rule.** All other identifier item trees are mapped to the empty program:

$$(tree, tm, expn) \mapsto ("", "", ""); \text{ else}$$

### 5.5.3 Assignment Interpretation of Hybel Item Trees

Some hybel item trees have different interpretations either for evaluation of their values or in the context of assignment. The latter is defined as a mapping

$$asscode : tree_o HybelItem \times ExpN \rightarrow String^3$$

As for *code*, there are main code, pre-code and post-code mappings for *asscode*:

$$assmain : tree_o HybelItem \times ExpN \rightarrow String$$

$$tn \mapsto \pi_1 asscode(tn)$$

$$asspre : tree_o HybelItem \times ExpN \rightarrow String$$

$$tn \mapsto \pi_2 asscode(tn)$$

$$asspost : tree_o HybelItem \times ExpN \rightarrow String$$

$$tn \mapsto \pi_3 asscode(tn)$$

**Variables.** The assignment interpretation of any variable is determined by the assignment interpretation of its identifier item tree:

$$(((t, r), idtree), sub), expn) \mapsto asscode_{Id}(idtree, expn)$$

$$\text{with } r \in \{var, derivVar\}$$

*Examples:* D.2.2, D.2.3, D.2.4, D.2.5, D.2.6, D.2.9, D.2.10, D.2.11.

**Sending of Signals.** Signal raise statements are composed of (1) the sending of the signal itself, which is defined by the assignment interpretation of the identifier item tree, and (2) the writing of the actual parameter values. The latter is defined here. A dedicated index variable is used to choose the signal's

index wrt. to its multiplicity for sending. It is assumed that this variable holds the appropriate value, prepared in the context of the raise statement.

$$\begin{aligned}
& (((sigtype, sendSig), idtree), \langle s_1, \dots, s_n \rangle), expn) \mapsto \\
& (assmain_{Id}(idtree, expn) \hat{ } \\
& \text{string}(lvar_{SigParam}(expn, s, write, 1)) \hat{ } "]" \\
& \hat{ } \text{string}(lvar_{sigIdx}(expn, s)) \hat{ } "]" \hat{ } ":" \hat{ } main(s_1, cur, expn) \hat{ } ";" ; \\
& \hat{ } \dots \hat{ } \text{repeat for } 1..n \\
& \text{string}(lvar_{SigParam}(expn, s, write, n)) \hat{ } "]" \\
& \hat{ } \text{string}(lvar_{sigIdx}(expn, s)) \hat{ } "]" \hat{ } ":" \hat{ } main(s_n, cur, expn) \hat{ } ";" ; , \\
& asspre_{Id}(idtree, expn) \hat{ } pre(s_1, cur, expn) \hat{ } \dots \hat{ } pre(s_n, cur, expn), \\
& asspost_{Id}(idtree, expn) \hat{ } post(s_1, cur, expn) \hat{ } \dots \hat{ } post(s_1, cur, expn)) \\
& \text{with } idtree = ((sendSig, (s, acc)), sub_{id})
\end{aligned}$$

**Receiving of Signals.** The receiving of signals (defined by trigger expressions) leads to the assignment of local variables from the parameter values that are associated with the signal's occurrence. The handling of the signal itself is defined later by a rule for the included identifier item tree.

$$\begin{aligned}
& (((sigtype, recvSig), idtree), \langle s_1, \dots, s_n \rangle), expn) \mapsto \\
& (assmain_{Id}(idtree, expn) \hat{ } \\
& assmain_{Id}(s_1, expn) \hat{ } ":" \hat{ } \text{string}(lvar_{SigParam}(expn, s, read, 1)) \\
& \hat{ } "]" \hat{ } \text{string}(lvar_{trgIdx}(expn, s)) \hat{ } "]" ; \\
& \hat{ } \dots \hat{ } \text{repeat for } 1..n \\
& assmain_{Id}(s_n, expn) \hat{ } ":" \hat{ } \text{string}(lvar_{SigParam}(expn, s, read, n)) \\
& \hat{ } "]" \hat{ } \text{string}(lvar_{trgIdx}(expn, s)) \hat{ } "]" ; , \\
& asspre_{Id}(idtree, expn) \hat{ } asspre_{Id}(s_1, expn) \hat{ } \dots \hat{ } asspre_{Id}(s_n, expn), \\
& asspost_{Id}(idtree, expn) \hat{ } asspost_{Id}(s_1, expn) \hat{ } \dots \hat{ } asspost_{Id}(s_n, expn)) \\
& \text{with } idtree = ((recvSig, (s, acc)), sub_{id})
\end{aligned}$$

**Assignment Expressions.** The assignment interpretation of simple assignments is almost identical with the expression interpretation; the only difference is that the resulting main code forms a program *statement*, rather than a program *expression*:

$$\begin{aligned}
& (((t, r), val), sub), expn) \mapsto \\
& \text{main}(((t, r), val), sub, cur, expn) \hat{ } ";" ; , \\
& \text{pre}(((t, r), val), sub, cur, expn), \\
& \text{post}(((t, r), val), sub, cur, expn))
\end{aligned}$$

with  $r \in \{ass, diffAss\}$

*Examples:* D.2.3, D.2.4, D.2.9, D.2.11.

**Assignment Group Expressions.** A group of assignments is transformed into a set of loops that iterate all specified values for the bound integer variable and always execute the bound assignment with the respective value:

$$\begin{aligned}
& (((t, r), idtree_{bnd}), \langle (((intSet, intSpecs), val), \\
& \quad \langle (((intSet, intRange), val_1), \langle s_{low,1}, s_{up,1} \rangle), \dots, \\
& \quad \langle (((intSet, intRange), val_n), \langle s_{low,n}, s_{up,n} \rangle) \rangle \rangle), boundExp \rangle), expn) \mapsto \\
& string(lvar_{V, V_{local}}(expn, v_{bnd})) \wedge "[0] := " \wedge main(s_{low,1}, cur, expn) \wedge "; " \\
& \wedge "while((" \wedge string(lvar_{V, V_{local}}(expn, v_{bnd})) \wedge "[0] \leq " \\
& \wedge main(s_{up,1}, cur, expn) \wedge ") \{ " \\
& \wedge assmain(boundExp, expn) \wedge \\
& "++" \wedge string(lvar_{V, V_{local}}(expn, v_{bnd})) \wedge "[0]; \} " \\
& \wedge \dots \wedge \text{repeat for } 1..n \\
& string(lvar_{V, V_{local}}(expn, v_{bnd})) \wedge "[0] := " \wedge main(s_{low,n}, cur, expn) \wedge "; " \\
& \wedge "while((" \wedge string(lvar_{V, V_{local}}(expn, v_{bnd})) \wedge "[0] \leq " \\
& \wedge main(s_{up,n}, cur, expn) \wedge ") \{ " \\
& \wedge assmain(boundExp, expn) \wedge \\
& "++" \wedge string(lvar_{V, V_{local}}(expn, v_{bnd})) \wedge "[0]; \} ", \\
& pre(s_{low,1}, cur, expn) \wedge pre(s_{up,1}, cur, expn) \wedge asspre(boundExp, expn) \\
& \wedge \dots \wedge \text{repeat for } 1..n \\
& pre(s_{low,n}, cur, expn) \wedge pre(s_{up,n}, cur, expn) \wedge asspre(boundExp, expn), \\
& post(s_{low,1}, cur, expn) \wedge post(s_{up,1}, cur, expn) \wedge asspost(boundExp, expn) \\
& \wedge \dots \wedge \text{repeat for } 1..n \\
& post(s_{low,n}, cur, expn) \wedge post(s_{up,n}, cur, expn) \wedge asspost(boundExp, expn)) \\
& \text{with } idtree_{bnd} = ((int, (v_{bnd}, acc)), sub_{id}) \wedge r \in \{assGroup, diffAssGroup\}
\end{aligned}$$

*Examples:* D.2.1, D.2.2, D.2.5, D.2.6.

**Non-Deterministic Integer Assignment Expressions.** A non-deterministic integer assignment is mapped to a program that iterates all given specified integer values, implemented by a set of loops. For each value, the bound boolean expression is evaluated, and the corresponding integer value is stored, if satisfied. For this, an array variable that stores the satisfied integers (given by  $lvar_{hits}$ ) and a size counter (from mapping  $lvar_{hitCount}$ ) are used. Finally, from the integers with satisfied expression evaluation, a random one is chosen.

$$(assTree, expn) \mapsto$$

```

(string(lvarhitCount(expn, assTree)) ^ ":= 0;" ^
string(lvarV, Vlocal(expn, vbnd) ^ "[0] :=" ^ main(slow,1, cur, expn) ^ ";" ^
^ "while((" ^ string(lvarV, Vlocal(expn, vbnd) ^ "[0] ≤"
^ main(sup,1, cur, expn) ^ ") ^ ") ^ (" ^
string(lvarhitCount(expn, assTree) ^ "<" ^ string(maxHitCount) ^ ")") {" ^
"if(" ^ main(boundExp, cur, expn) ^ ") {" ^
string(lvarhits(expn, assTree))
^ "[" ^ string(lvarhitCount(expn, assTree)) ^ "]" :="
^ string(lvarV, Vlocal(expn, vbnd) ^ "[0];" ^
"++" ^ string(lvarhitCount(expn, assTree)) ^ ";" ^
"}" ^
"++" ^ string(lvarV, Vlocal(expn, vbnd) ^ "[0];" ^
"}" ^
^ ... ^ repeat for 1..n
string(lvarV, Vlocal(expn, vbnd) ^ "[0] :=" ^ main(slow,n, cur, expn) ^ ";" ^
^ "while((" ^ string(lvarV, Vlocal(expn, vbnd) ^ "[0] ≤"
^ main(sup,n, cur, expn) ^ ") ^ ") ^ (" ^
string(lvarhitCount(expn, assTree) ^ "<" ^ string(maxHitCount) ^ ")") {" ^
"if(" ^ main(boundExp, cur, expn) ^ ") {" ^
string(lvarhits(expn, assTree))
^ "[" ^ string(lvarhitCount(expn, assTree)) ^ "]" :="
^ string(lvarV, Vlocal(expn, vbnd) ^ "[0];" ^
"++" ^ string(lvarhitCount(expn, assTree)) ^ ";" ^
"}" ^
"++" ^ string(lvarV, Vlocal(expn, vbnd) ^ "[0];" ^
"}" ^
"if(" ^ string(lvarhitCount(expn, assTree)) ^ "> 0) {" ^
assmain(varExp, expn) ^ ":=" ^ string(lvarhits(expn, assTree)) ^ "[" ^
"random() mod" ^ string(lvarhitCount(expn, assTree)) ^ ";" ^
"}",
pre(slow,1, cur, expn) ^ pre(sup,1, cur, expn) ^ pre(boundExp, cur, expn)
^ ... ^ repeat for 1..n
pre(slow,n, cur, expn) ^ pre(sup,n, cur, expn) ^ pre(boundExp, cur, expn)
^ asspre(varExp, expn)
post(slow,1, cur, expn) ^ post(sup,1, cur, expn) ^ post(boundExp, cur, expn)
^ ... ^ repeat for 1..n
post(slow,n, cur, expn) ^ post(sup,n, cur, expn) ^ post(boundExp, cur, expn)
^ asspost(varExp, expn)

```

with  $assTree = (((int, intNondetAss), ((int, (v_{bnd}, acc)), sub_{id})),$   
 $\langle varExp, (((intSet, intSpecs), val), (((intSet, intRange), val_1),$   
 $\langle s_{low,1}, s_{up,1} \rangle)), \dots,$   
 $((intSet, intRange), val_n), \langle s_{low,n}, s_{up,n} \rangle \rangle \rangle, boundExp))$

Note that there is a defined maximum number  $maxHitCount$  of integers that can be stored intermediately, such that the non-determinism is restricted to the first  $maxHitCount$  integers for which the bound expression is satisfied.

*Examples:* D.2.10.

**Default Rule.** For all hybel item trees that have no assignment interpretation, the empty program is created:

$$(tree, expn) \mapsto ("", "", ""); \text{ else}$$

### 5.5.4 Assignment Interpretation of Identifier Item Trees

Corresponding to hybel item trees, identifier item trees have an assignment interpretation. There is a mapping to program strings, which also depends on the expression node for which the program is created.

$$asscode_{Id} : tree_o IdItem \times ExpN \rightarrow String^3$$

There are main code, pre-code and post-code projections, given by the mappings

$$assmain_{Id} : tree_o IdItem \times ExpN \rightarrow String$$

$$tn \mapsto \pi_1 asscode_{Id}(tn)$$

$$asspre_{Id} : tree_o IdItem \times ExpN \rightarrow String$$

$$tn \mapsto \pi_2 asscode_{Id}(tn)$$

$$asspost_{Id} : tree_o IdItem \times ExpN \rightarrow String$$

$$tn \mapsto \pi_3 asscode_{Id}(tn)$$

**Variable Identifiers.** Variable identifiers are mapped to the local array variable that represents *write* access to the HybridUML variable or signal. The array index results from the index expression:

$$(((t, (v, acc)), ((indexExp, exptree), \langle \rangle)), expn) \mapsto$$

$$(string(lvar_{V, V_{local}}(expn, v, write, cur))$$

$$\wedge "[]" \wedge main_{Id}(((indexExp, exptree), \langle \rangle), cur, expn) \wedge "]",$$

$$pre_{Id}(((indexExp, exptree), \langle \rangle), cur, expn),$$

$$post_{Id}(((indexExp, exptree), \langle \rangle), cur, expn))$$

with  $t \in DT$

Variable identifiers without index expression are treated similarly, array index 0 is applied.

$$(((t, (v, acc)), \langle \rangle), expn) \mapsto$$

$$(string(lvar_{V, V_{local}}(expn, v, write, cur)) \wedge "[0]", "", ""))$$

with  $t \in DT$

*Examples:* D.2.2, D.2.3, D.2.4, D.2.5, D.2.6, D.2.9, D.2.10, D.2.11.

**Sub-Variables of Structured Data Types.** Access to a sub-variable of a variable of structured data type is defined recursively, in the same fashion as for  $code_{Id}$ .

$$\begin{aligned}
&(((t, (v, acc)), \langle s_1, s_2 \rangle), expn) \mapsto \\
&(string(lvar_{V, V_{local}}(expn, v, write, cur)) \hat{\ } " [ " \hat{\ } main_{Id}(s_1, cur, expn) \hat{\ } " ] . " \\
&\hat{\ } \hat{\ } assmain_{Id}(s_2, expn), \\
&pre_{Id}(s_1, cur, expn) \hat{\ } \hat{\ } asspre_{Id}(s_2, expn), \\
&post_{Id}(s_1, cur, expn) \hat{\ } \hat{\ } asspost_{Id}(s_2, expn)) \\
&\text{with } t \in DT_{struc}
\end{aligned}$$

Without index expression, index 0 is applied:

$$\begin{aligned}
&(((t, (v, acc)), \langle \langle (t_s, val_s), sub_s \rangle \rangle), expn) \mapsto \\
&(string(lvar_{V, V_{local}}(expn, v, write, cur)) \hat{\ } " [ 0 ] . " \\
&\hat{\ } \hat{\ } assmain_{Id}(\langle \langle (t_s, val_s), sub_s \rangle \rangle, expn), \\
&asspre_{Id}(\langle \langle (t_s, val_s), sub_s \rangle \rangle, expn), \\
&asspost_{Id}(\langle \langle (t_s, val_s), sub_s \rangle \rangle, expn)) \\
&\text{with } t \in DT_{struc} \wedge t_s \neq indexExp
\end{aligned}$$

**Signal Identifiers on Signal Reception.** The assignment interpretation of triggers is the consumption of the signal's occurrence. This is done by publishing the value *false* for the signal. Here, the value is written to the corresponding local variable. For a conventional index expression, the affected trigger index is given by the index expression; a dedicated trigger index variable is set to this index as a side-effect. This is supposed to be used by special code for the publication of the value, which is defined in section 5.5.7.

$$\begin{aligned}
&(((recvSig, (s, recv)), \langle \langle (indexExp, (\langle (int, r), val_1 \rangle), sub_1 \rangle), \langle \rangle \rangle \rangle), expn) \mapsto \\
&(string(lvar_{trgIdx}(expn, s)) \hat{\ } " := " \\
&\hat{\ } \hat{\ } main(\langle \langle (int, r), val_1 \rangle \rangle, sub_1, cur, expn) \hat{\ } " ; " \hat{\ } \hat{\ } \\
&(string(lvar_S(expn, s, write)) \\
&\hat{\ } \hat{\ } " [ " \hat{\ } string(lvar_{trgIdx}(expn, s)) \hat{\ } " ] := false; ", \\
&pre(\langle \langle (int, r), val_1 \rangle \rangle, sub_1, cur, expn), \\
&post(\langle \langle (int, r), val_1 \rangle \rangle, sub_1, cur, expn)) \\
&\text{with } r \neq indexAss
\end{aligned}$$

Without index expression, index 0 is used:

$$\begin{aligned}
&(((recvSig, (s, recv)), \langle \rangle), expn) \mapsto \\
&string(lvar_{trgIdx}(expn, s)) \hat{\ } " := 0; " \hat{\ } \hat{\ } \\
&(string(lvar_S(expn, s, write)) \\
&\hat{\ } \hat{\ } " [ " \hat{\ } string(lvar_{trgIdx}(expn, s)) \hat{\ } " ] := false; ", \\
&"", ""))
\end{aligned}$$

On reception of a signal with index assignment expression, all indices of the signal have to be iterated, in order find an affected one. The first index for which the signal is active is taken, such that for several indices, the smallest wins:

```

(((recvSig, (s, recv)),
  (((indexExp, (((int, indexAss), val1), ⟨sass))), ⟨⟩)), expn) ↦

(string(lvartrgIdx(expn, s)) ^ " := 0; " ^
"while ((^ string(lvartrgIdx(expn, s)) ^ "<" ^ string(n)
^ ") ^ (¬ ^ string(lvarS(expn, s, read))
^ "[ ^ string(lvartrgIdx(expn, s)) ^ "]") { ^
"++" ^ string(lvartrgIdx(expn, s)) ^ "; " ^
"}" ^
assmain(sass, expn) ^ " := " ^ string(lvartrgIdx(expn, s)) ^ "; " ^
string(lvarS(expn, s, write)) ^ "[ ^ string(lvartrgIdx(expn, s)) ^ "]"
^ " := false; ",

asspre(sass, expn),

asspost(sass, expn))

with n = |snexpn(s, expn)| - 1

```

**Signal Identifiers on Signal Sending.** Sending of signals is realized by programs that write the value *true* to the corresponding local variable, such that this will be published to the signal's channel. A dedicated index variable holds the index value for this reason.

```

(((sendSig, (s, send)), ⟨sidx⟩), expn) ↦

(string(lvarsigIdx(expn, s)) ^ " := " ^ mainId(sidx, cur, expn) ^ "; " ^
string(lvarS(expn, s, write)) ^ "[ ^ string(lvarsigIdx(expn, s)) ^ "]"
^ " := true; ",

preId(sidx, cur, expn),

preId(sidx, cur, expn))

```

Without an index expression, index 0 is used.

```

(((sendSig, (s, send)), ⟨⟩), expn) ↦

(string(lvarsigIdx(expn, s)) ^ " := 0; " ^
string(lvarS(expn, s, write)) ^ "[ ^ string(lvarsigIdx(expn, s)) ^ "]"
^ " := true; ",

" ", " ")

```

**Conventional Index Expressions.** Conventional index expressions of identifier items are given by hybel item trees:

$$(((indexExp, exptree), sub), expn) \mapsto asscode(exptree, expn)$$

*Examples:* D.2.1, D.2.2, D.2.5, D.2.6, D.2.10.

**Default Rule.** The assignment interpretation of all remaining identifier item trees is the empty program:

$$(tree, expn) \mapsto ("", "", ""); \text{ else}$$

### 5.5.5 Initialization of Local Variables from Channels

Before any program code is executed that implements the evaluation or assignment interpretation of expression nodes, dedicated local variables must be initialized. That are the local variables which provide read access to values that are read from channels. All other local variables are initialized inside the respective program code.

Therefore, the mapping  $initcode : \mathcal{P}(ExpN) \rightarrow String^6$  provides the initialization code, such that the projections define the initialization of the following local variables:

1. The initialization code for local variables reading the current value of a HybridUML variable from a channel is given by

$$\begin{aligned} initcode_{V,cur} : \mathcal{P}(ExpN) &\rightarrow String \\ expn &\mapsto \pi_1 initcode(expn) \end{aligned}$$

2. Local variables reading the previous value of a HybridUML variable are initialized by the code

$$\begin{aligned} initcode_{V,prev} : \mathcal{P}(ExpN) &\rightarrow String \\ expn &\mapsto \pi_2 initcode(expn) \end{aligned}$$

3. The time stamp for the current value of a HybridUML variable is read and assigned by the code

$$\begin{aligned} initcode_{V,curTck} : \mathcal{P}(ExpN) &\rightarrow String \\ expn &\mapsto \pi_3 initcode(expn) \end{aligned}$$

4. The previous time stamp is initialized with

$$\begin{aligned} initcode_{V,prevTck} : \mathcal{P}(ExpN) &\rightarrow String \\ expn &\mapsto \pi_4 initcode(expn) \end{aligned}$$

5. Initializing variables holding the enabled-flags of HybridUML signals is defined as

$$\begin{aligned} initcode_S : \mathcal{P}(ExpN) &\rightarrow String \\ expn &\mapsto \pi_5 initcode(expn) \end{aligned}$$

6. The initialization of signal parameters is given by

$$\begin{aligned} \text{initcode}_{\text{SigParam}} &: \mathcal{P}(\text{ExpN}) \rightarrow \text{String} \\ \text{expn} &\mapsto \pi_6 \text{initcode}(\text{expn}) \end{aligned}$$

The initialization code assignment is then defined by a collection of several mappings. Sets of expression nodes are mapped to initialization code for the hybrid trees that represent their expressions:

$$\begin{aligned} \text{initcode} &: \mathcal{P}(\text{ExpN}) \rightarrow \text{String}^6 \\ \text{expnset} &\mapsto \text{initcode}_1(\{(var_{ht,read}(ht_{\text{expn}}(\text{expn})), \text{expn}) \mid \\ &\quad \text{expn} \in \text{expnset} \cap \text{ExpN}_{\text{spec}}\} \\ &\quad \cup \{(sig_{ht,read}(ht_{\text{expn}}(\text{expn})), \text{expn}) \mid \text{expn} \in \text{expnset} \cap \text{ExpN}_{\text{spec}}\}) \end{aligned}$$

This is defined by the initialization of sets of variables and signals, within expression nodes:

$$\begin{aligned} \text{initcode}_1 &: \mathcal{P}(\mathcal{P}(V \cup S) \times \text{ExpN}_{\text{spec}}) \rightarrow \text{String}^6 \\ \text{vse} &\mapsto \text{initcode}_2(\{(item, \text{expn}) \mid \exists (vs, \text{expn}) \in \text{vse} \bullet item \in vs\}) \end{aligned}$$

Initialization of pairs of items, i.e. variables or signals, and expression nodes, is defined separately for the local HL<sup>3</sup> variables listed above. Therefore, the local variables are determined from the HybridUML variables and signals, along with the expression node. For each local variable, the set of indices is chosen that corresponds to the multiplicity from the HybridUML specification. Each index is associated with a port that is used to read data from the associated channel:

$$\begin{aligned} \text{initcode}_2 &: \mathcal{P}((V \cup S) \times \text{ExpN}_{\text{spec}}) \rightarrow \text{String}^6 \\ \text{vse} &\mapsto ( \\ &\quad \text{initcode}_3(\{(\{(port_{VN_{\text{basic}}}(vn), index_{VN}(vn)) \mid vn \in vn_{\text{expn}}(v, \text{expn})\}, \\ &\quad \quad lvar_{V, V_{\text{local}}}(\text{expn}, v, read, cur), cur) \mid v \in V \wedge (v, \text{expn}) \in \text{vse}\}), \\ &\quad \text{initcode}_3(\{(\{(port_{VN_{\text{basic}}}(vn), index_{VN}(vn)) \mid vn \in vn_{\text{expn}}(v, \text{expn})\}, \\ &\quad \quad lvar_{V, V_{\text{local}}}(\text{expn}, v, read, prev), prev) \mid v \in V \wedge (v, \text{expn}) \in \text{vse}\}), \\ &\quad \text{initcode}_3(\{(\{(port_{VN_{\text{basic}}}(vn), index_{VN}(vn)) \mid vn \in vn_{\text{expn}}(v, \text{expn})\}, \\ &\quad \quad lvar_{V, curTck}(\text{expn}, v), curTck) \mid v \in V \wedge (v, \text{expn}) \in \text{vse}\}), \\ &\quad \text{initcode}_3(\{(\{(port_{VN_{\text{basic}}}(vn), index_{VN}(vn)) \mid vn \in vn_{\text{expn}}(v, \text{expn})\}, \\ &\quad \quad lvar_{V, prevTck}(\text{expn}, v), prevTck) \mid v \in V \wedge (v, \text{expn}) \in \text{vse}\}), \\ &\quad \text{initcode}_3(\{(\{(port_{SN_{\text{basic}}}(sn), index_{SN}(sn)) \mid sn \in sn_{\text{expn}}(s, \text{expn})\}, \\ &\quad \quad lvar_S(\text{expn}, s, read), cur) \mid s \in S \wedge (s, \text{expn}) \in \text{vse}\}), \\ &\quad \text{initcode}_3(\{(\{(port_{SN_{\text{basic}, param}}(sn, n), index_{SN}(sn)) \mid sn \in sn_{\text{expn}}(s, \text{expn})\}, \\ &\quad \quad lvar_{\text{SigParam}}(\text{expn}, s, read, n), cur) \mid \\ &\quad \quad s \in S \wedge (s, \text{expn}) \in \text{vse} \wedge 0 \leq n < |paramTypes_S(s)|\}) \end{aligned}$$

Initialization for sets of local variables with corresponding sets of ports and indices are given wrt. to the kind of data access from the channel:

$$\text{Acc}_{\text{initcode}} = \{cur, prev, curTck, prevTck\}$$

$$\begin{aligned}
\mathit{initcode}_3 &: \mathcal{P}(\mathcal{P}(\mathit{Port} \times \mathbb{N}_0) \times \mathit{Var}) \times \mathit{Acc}_{\mathit{initcode}} \rightarrow \mathit{String} \\
(\mathit{pilv}, \mathit{acc}) &\mapsto \mathit{initcode}_4(\mathit{anyseq}_{\mathit{seq}(\mathit{Port} \times \mathbb{N}_0) \times \mathit{Var}} \\
&\quad (\{(\mathit{anyseq}_{\mathit{Port} \times \mathbb{N}_0}(\mathit{pi}), \mathit{lv}) \mid (\mathit{pi}, \mathit{lv}) \in \mathit{pilv}\}), \mathit{acc})
\end{aligned}$$

The sets of local variables and the sets of variable and signal nodes are evaluated in an arbitrary order.<sup>2</sup>

The entries of a sequence of local variables with corresponding port and index is then transformed into code, recursively:

$$\begin{aligned}
\mathit{initcode}_4 &: \mathit{seq}(\mathit{seq}(\mathit{Port} \times \mathbb{N}_0) \times \mathit{Var}) \times \mathit{Acc}_{\mathit{initcode}} \rightarrow \mathit{String} \\
(\langle h \rangle \frown \mathit{sq}, \mathit{acc}) &\mapsto \mathit{initcode}_5(h, \mathit{acc}) \frown \mathit{initcode}_4(\mathit{sq}, \mathit{acc}) \\
(\langle \rangle, \mathit{acc}) &\mapsto ""
\end{aligned}$$

A single variable along with a sequence of ports and indices is mapped to a sequence of initialization statements, corresponding to the data access type:

$$\begin{aligned}
\mathit{initcode}_5 &: \mathit{seq}(\mathit{Port} \times \mathbb{N}_0) \times \mathit{Var} \times \mathit{Acc}_{\mathit{initcode}} \rightarrow \mathit{String} \\
(\langle (p, n) \rangle \frown \mathit{sq}, \mathit{lv}, \mathit{acc}) &\mapsto \mathit{string}(\mathit{lv}) \frown "[ " \frown \mathit{string}(n) \frown "]" := " \\
&\quad \frown \mathit{initcode}_6(\mathit{acc}) \frown "(" \frown \mathit{string}(p) \frown ")"; " \frown \mathit{initcode}_5(\mathit{sq}, \mathit{lv}, \mathit{acc}) \\
\langle \rangle &\mapsto ""
\end{aligned}$$

Here, the local variable is used directly, and the corresponding array index is taken from the property or signal node.

Finally, the program statement is determined from the data access type:

$$\begin{aligned}
\mathit{initcode}_6 &: \mathit{Acc}_{\mathit{initcode}} \rightarrow \mathit{String} \\
\mathit{cur} &\mapsto \text{"get"} \\
\mathit{prev} &\mapsto \text{"getPrevious"} \\
\mathit{curTck} &\mapsto \text{"getTime"} \\
\mathit{prevTck} &\mapsto \text{"getPreviousTime"}
\end{aligned}$$

*Examples:* All examples of section D.2 initialize their local variables at the beginning of the code. Note two technical differences here: (1) Initialization and declaration of (some) variables is merged for the C++ code. Further, there are additional wrapper variables for technical reasons, e.g. for bounds-safe array access. (2) Some of the variables are references to the corresponding port's data buffer, therefore modification of these variables directly accesses the respective buffer. For this reason, the port's send statements do not explicitly contain the local variable, but use the port's data buffer by default.

### 5.5.6 Visibility Set Parameter Access

In addition to the initialization of local variables from channels, for some programs, a visibility set parameter is provided. This is accessed by the special HL<sup>3</sup> statement `getVisParam`, such that it is stored in dedicated local variables.

<sup>2</sup>The mapping  $\mathit{anyseq}_X : \mathcal{P}(X) \rightarrow \mathit{seq} X$  is supposed to define an arbitrary sequence for all elements of a given set  $x \subseteq X$ , i.e.:  $\forall x \subseteq X \bullet \mathit{ran}(\mathit{anyseq}_X(x)) = x \wedge |\mathit{anyseq}_X(x)| = |x|$

Therefore, for a set of expression nodes, the corresponding variables are collected as a sequence, but omitting duplicates:

$$\begin{aligned}
& \text{visparam}_{\text{seq } \text{ExpN}} : \text{seq } \text{ExpN} \rightarrow \text{seq } \text{Var} \\
& \langle \rangle \mapsto \langle \rangle \\
& \langle \text{expn} \rangle \wedge \text{sq} \mapsto \text{lvar}_{\text{visSet}}(\text{expn}) \wedge \text{visparam}_{\text{seq } \text{ExpN}}(\text{sq}) \\
& \quad ; \text{expn} \in \text{dom } \text{lvar}_{\text{visSet}} \wedge \text{lvar}_{\text{visSet}}(\text{expn}) \notin \text{visparam}_{\text{seq } \text{ExpN}}(\text{sq}) \\
& \langle \text{expn} \rangle \wedge \text{sq} \mapsto \text{visparam}_{\text{seq } \text{ExpN}}(\text{sq}); \text{ else}
\end{aligned}$$

From a sequence of such variables, a program string of initializations is created:

$$\begin{aligned}
& \text{visparamcode}_{\text{seq } \text{Var}} : \text{seq } \text{Var} \rightarrow \text{String} \\
& \langle \text{var} \rangle \wedge \text{sq} \mapsto \text{string}(\text{var}) \wedge \text{" := getVisParam();"} \wedge \text{visparamcode}_{\text{seq } \text{Var}}(\text{sq}) \\
& \langle \rangle \mapsto \langle \rangle
\end{aligned}$$

Then, initialization code for expression node sets is given:

$$\begin{aligned}
& \text{initcode}_{\text{visparam}} : \mathcal{P}(\text{ExpN}) \mapsto \text{String} \\
& \text{expnset} \mapsto \text{visparamcode}_{\text{seq } \text{Var}}(\text{visparam}_{\text{seq } \text{ExpN}}(\text{anyseq}_{\text{ExpN}}(\text{expnset})))
\end{aligned}$$

*Examples:* C++ provides access to operation parameters implicitly. Of course, this is exploited for the generated code: D.2.1, D.2.2, D.2.3, D.2.4, D.2.5, D.2.6, D.2.9, D.2.10, D.2.11.

### 5.5.7 Publication of Local Variable Data to Channels

After program code is executed that implements the evaluation or assignment interpretation of expression nodes, the calculation results are contained in local variables. In order to publish the values, the variables' contents have to be written to channels, via corresponding ports.

The mapping  $\text{pubcode} : \mathcal{P}(\text{ExpN}) \rightarrow \text{String}^4$  defines the publication code, such that the projections define the publication of the following values:

1. The publication code for local variables containing a new value of a written HybridUML variable is given by

$$\begin{aligned}
& \text{pubcode}_V : \mathcal{P}(\text{ExpN}) \rightarrow \text{String} \\
& \text{expn} \mapsto \pi_1 \text{pubcode}(\text{expn})
\end{aligned}$$

2. Publication of newly raised HybridUML signal (excluding its parameters) is defined as

$$\begin{aligned}
& \text{pubcode}_{S,\text{send}} : \mathcal{P}(\text{ExpN}) \rightarrow \text{String} \\
& \text{expn} \mapsto \pi_2 \text{pubcode}(\text{expn})
\end{aligned}$$

3. The publication of parameters of a raised signal is given by

$$\begin{aligned}
& \text{pubcode}_{\text{SigParam}} : \mathcal{P}(\text{ExpN}) \rightarrow \text{String} \\
& \text{expn} \mapsto \pi_3 \text{pubcode}(\text{expn})
\end{aligned}$$

4. Publication of the consumption of a trigger only affects the signal channel itself and is defined as

$$\begin{aligned} \text{pubcode}_{S,recv} : \mathcal{P}(\text{ExpN}) &\rightarrow \text{String} \\ \text{expn} &\mapsto \pi_4 \text{pubcode}(\text{expn}) \end{aligned}$$

In the same fashion as the initialization code was defined in section 5.5.5, the publication code is defined by a collection of mappings. Sets of expression nodes are mapped to the publication code, corresponding to the hybel trees that represent the contained expressions:

$$\begin{aligned} \text{pubcode} : \mathcal{P}(\text{ExpN}) &\rightarrow \text{String}^4 \\ \text{expnset} &\mapsto \text{pubcode}_1(\{(var_{ht,write}(ht_{expn}(\text{expn})), \text{expn}) \mid \\ &\quad \text{expn} \in \text{expnset} \cap \text{ExpN}_{spec}\} \\ &\quad \cup \{(sig_{ht,write}(ht_{expn}(\text{expn})), \text{expn}) \mid \text{expn} \in \text{expnset} \cap \text{ExpN}_{spec}\} \\ &\quad \cup \{(sig_{ht,read}(ht_{expn}(\text{expn})), \text{expn}) \mid \text{expn} \in \text{expnset} \cap \text{ExpN}_{spec}\}) \end{aligned}$$

This is given by the publication code for sets of variables and signals, within expression nodes:

$$\begin{aligned} \text{pubcode}_1 : \mathcal{P}(\mathcal{P}(V \cup S) \times \text{ExpN}_{spec}) &\rightarrow \text{String}^4 \\ \text{vse} &\mapsto \text{pubcode}_2(\{(item, \text{expn}) \mid \exists (vs, \text{expn}) \in \text{vse} \bullet item \in vs\}) \end{aligned}$$

For the sets of HybridUML variables and signals within an expression node, publication code is distinguished as described above, i.e. sending of (1) written variables, (2) raised signals, (3) the corresponding signal parameters, and (4) consumption of signals are handled separately.

The local variables which hold the new values are determined from the HybridUML variables and signals, along with the expression node. For each local variable, the set of indices corresponding to the multiplicity is created. The ports for writing are determined and attached to the respective indices.

$$\begin{aligned} \text{pubcode}_2 : \mathcal{P}((V \cup S) \times \text{ExpN}_{spec}) &\rightarrow \text{String}^4 \\ \text{vse} &\mapsto ( \\ &\quad \text{pubcode}_{3,V}(\{(port_{VN_{basic}}(vn), index_{VN}(vn)) \mid vn \in vn_{expn}(v, \text{expn})\}, \\ &\quad \quad \text{lvar}_{V,V_{local}}(\text{expn}, v, read, cur), \text{lvar}_{visSet}(\text{expn})) \mid \\ &\quad \quad v \in V \wedge (v, \text{expn}) \in \text{vse}\}), \\ &\quad \text{pubcode}_{3,S,write}(\{(port_{SN_{basic}}(sn), index_{SN}(sn)) \mid sn \in sn_{expn}(s, \text{expn})\}, \\ &\quad \quad \text{lvar}_S(\text{expn}, s, write), \text{lvar}_{visSet}(\text{expn}), \text{lvar}_{sigIdx}(\text{expn}, s)) \mid \\ &\quad \quad s \in S \wedge (s, \text{expn}) \in \text{vse}\}), \\ &\quad \text{pubcode}_{3,S,write}(\{(port_{SN_{basic,param}}(sn, n), index_{SN}(sn)) \mid \\ &\quad \quad sn \in sn_{expn}(s, \text{expn})\}, \\ &\quad \quad \text{lvar}_{SigParam}(\text{expn}, s, write, n), \text{lvar}_{visSet}(\text{expn}), \text{lvar}_{sigIdx}(\text{expn}, s)) \mid \\ &\quad \quad s \in S \wedge (s, \text{expn}) \in \text{vse} \wedge 0 \leq n < |paramTypes_S(s)|\}), \\ &\quad \text{pubcode}_{3,S,read0}(\{\text{expn} \mid \exists s \in S \bullet (s, \text{expn}) \in \text{vse}\}) \frown \\ &\quad \quad \text{pubcode}_{3,S,read}(\{(port_{SN_{basic}}(sn), index_{SN}(sn)) \mid \\ &\quad \quad sn \in sn_{expn}(s, \text{expn})\}, \end{aligned}$$

$$lvar_S(expn, s, read), lvar_{visSet}(expn), lvar_{trgIdx}(expn, s), lvar_{vis}(expn) \\ | s \in S \wedge (s, expn) \in use\})$$

For the respective kinds of values, the sets of variables with attached ports and indices are mapped to sequences, and the variables are iterated, in the same fashion as it was done for the initialization code:

$$\begin{aligned} &pubcode_{3,V} : \mathcal{P}(\mathcal{P}(Port \times \mathbb{N}_0) \times Var \times Var) \rightarrow String \\ &v \mapsto pubcode_{4,V}(\text{anyseq}_{seq(Port \times \mathbb{N}_0) \times Var \times Var} \\ &\quad (\{(anyseq_{Port \times \mathbb{N}_0}(pi), lv, visSet) \mid (pi, lv, visSet) \in v\})) \\ &pubcode_{3,S,write} : \mathcal{P}(\mathcal{P}(Port \times \mathbb{N}_0) \times Var \times Var \times Var) \rightarrow String \\ &v \mapsto pubcode_{4,S,write}(\text{anyseq}_{seq(Port \times \mathbb{N}_0) \times Var \times Var \times Var} \\ &\quad (\{(anyseq_{Port \times \mathbb{N}_0}(pi), lv, visSet, sigIdx) \mid (pi, lv, visSet, sigIdx) \in v\})) \\ &pubcode_{3,S,read} : \mathcal{P}(\mathcal{P}(Port \times \mathbb{N}_0) \times Var \times Var \times Var \times Var) \rightarrow String \\ &v \mapsto pubcode_{4,S,read}(\text{anyseq}_{seq(Port \times \mathbb{N}_0) \times Var \times Var \times Var \times Var} \\ &\quad (\{(anyseq_{Port \times \mathbb{N}_0}(pi), lv, visSet, sigIdx, vis) \mid \\ &\quad (pi, lv, visSet, sigIdx, vis) \in v\})) \\ \\ &pubcode_{4,V} : seq(seq(Port \times \mathbb{N}_0) \times Var \times Var) \rightarrow String \\ &\langle h \rangle \frown sq \mapsto pubcode_{5,V}(h) \frown pubcode_{4,V}(sq) \\ &\langle \rangle \mapsto "" \\ &pubcode_{4,S,write} : seq(seq(Port \times \mathbb{N}_0) \times Var \times Var \times Var) \rightarrow String \\ &\langle h \rangle \frown sq \mapsto pubcode_{5,S,write}(h) \frown pubcode_{4,S,write}(sq) \\ &\langle \rangle \mapsto "" \\ &pubcode_{4,S,read} : seq(seq(Port \times \mathbb{N}_0) \times Var \times Var \times Var \times Var) \\ &\quad \rightarrow String \\ &\langle h \rangle \frown sq \mapsto pubcode_{5,S,read}(h) \frown pubcode_{4,S,read}(sq) \\ &\langle \rangle \mapsto "" \end{aligned}$$

A single variable along with a sequence of ports and indices is mapped to a sequence of publication statements, given by mappings that correspond to the kind of value. Values for HybridUML variables are simply written to the channel. Besides the variable  $lv$  that holds the new value, the visibility set  $visSet$  is available, and used directly.

$$\begin{aligned} &pubcode_{5,V} : seq(Port \times \mathbb{N}_0) \times Var \times Var \rightarrow String \\ &(\langle (p, n) \rangle \frown sq, lv, visSet) \mapsto \text{put}(\text{string}(p) \frown \text{string}(visSet) \frown \text{string}(lv) \frown \text{string}(n)); \text{string}(n) \frown sq, lv, visSet) \\ &\langle \rangle, sq, lv, visSet \mapsto "" \end{aligned}$$

Raised signals, as well as signal parameters, are only written if the signal index coincides with the special local signal index variable's value, which is calculated by the code that precedes this publication code. Therefore, it is wrapped by a conditional statement. The available variables are  $lv$  and  $visSet$  as above, and

the signal index variable  $sigIdx$  for the comparison.

```

pubcode5,S,write : seq(Port ×  $\mathbb{N}_0$ ) × Var × Var × Var → String
((p, n) ^ sq, lv, visSet, sigIdx) ↦
  "if(" ^ string(sigIdx) ^ "==" ^ string(n) ^ "){" ^
  "put(" ^ string(p) ^ "," ^ string(visSet) ^ "," ^
  ^ string(lv) ^ "[" ^ string(n) ^ ");" ^
  ^ "}" ^ pubcode5,S,write((sq, lv, visSet, sigIdx)
(l), sq, lv, visSet, sigIdx) ↦ ""

```

Triggers are only consumed, if they are actually received; therefore, the signal index is compared with the special trigger index variable, similarly to the raising of signals, and therefore embedded into a conditional statement. Within, the visibility set for the publication is adapted such that only the sending port itself acts as recipient, such that the consumption of the signal is only locally effective. Therefore, the signal will remain active for all other abstract machines. The variables that are used here are  $lv$  and  $visSet$  as before, the trigger index variable  $trgIdx$  for the index comparison, and the visibility variable  $vis$ , which is used to define the local visibility of the publication.

```

pubcode5,S,read : seq(Port ×  $\mathbb{N}_0$ ) × Var × Var × Var × Var → String
((p, n) ^ sq, lv, visSet, trgIdx, vis) ↦
  "if(" ^ string(trgIdx) ^ "==" ^ string(n) ^ "){" ^
  "setRight(" ^ string(vis) ^ "," ^ string(p) ^ ");" ^
  "clear(" ^ string(visSet) ^ ");" ^
  "addEntry(" ^ string(visSet) ^ "," ^ string(vis) ^ ");" ^
  "put(" ^ string(p) ^ "," ^ string(visSet) ^ "," ^
  ^ string(lv) ^ "[" ^ string(n) ^ ");" ^
  ^ "}" ^ pubcode5,S,read((sq, lv, visSet, trgIdx, vis)
(l), sq, lv, visSet, trgIdx, vis) ↦ ""

```

As a prerequisite for the calculation of the visibility set above, the mapping  $pubcode_{4,S,read0}$  defines code to precede all conditional statements for signal consumption, that prepares the publication time and visibility  $vis$  for the value's publication. In order to iterate the affected expression nodes beforehand, the mapping  $pubcode_{3,S,read0}$  provides a sequence for this.

```

pubcode3,S,read0 :  $\mathcal{P}(ExpN_{spec})$  → String
expnset ↦ pubcode4,S,read0(anyseqExpNspec(expnset))

pubcode4,S,read0 : seq ExpNspec → String
<expn> ^ sq ↦
  string(lvarnewTck(expn)) ^ "==" ^ "getTime();" ^
  "setRight(" ^ string(lvarnewTck(expn)) ^ ",right(" ^
  ^ string(lvarnewTck(expn)) ^ ") + 1;" ^
  "setLeft(" ^ string(lvarvis(expn)) ^ ");"

```

$$\langle \rangle \mapsto \text{""}$$

$$\widehat{\text{string}}(lvar_{newTck}(expn)) \widehat{\text{""}};$$

$$\widehat{\text{pubcode}}_{A,S,read0}(sq)$$

*Examples:* D.2.1, D.2.2, D.2.3, D.2.4, D.2.5, D.2.6, D.2.9, D.2.10, D.2.11. Note that the port's send statements do not explicitly contain a reference to a local variable, because they use a port's internal data buffer. Local variables for write access are defined as references to the corresponding buffer on initialization.

## 5.6 HybridUML Abstract Subject Execution

This section provides the effects of the operations of abstract subjects. In section 4.5, HL<sup>3</sup> operational rules were defined for the execution of abstract subjects. For each of these rules, a corresponding *effect function* was declared, but not defined, because their definitions depend on the high-level formalism for which an HL<sup>3</sup> model is created. Therefore, the definitions of the effect functions given here are specific to HybridUML.

There are two kinds of abstract subjects – (1) Abstract Machines and (2) the Selector. Correspondingly, the definitions are structured into sections 5.6.1 and 5.6.2.

The operations' definitions rely on some *internal state* of abstract subjects, which is also specific for the high-level formalism HybridUML. Internal state is different for abstract machines and the selector, therefore the HybridUML internal state is the union of both:

$$IntState = IntState_{Am} \cup IntState_{sel}$$

### 5.6.1 Abstract Machines

The sequential behavior of an abstract machine  $am \in Am$  is defined on the basis of the tree of mode instance nodes of the associated agent instance node  $mtree_{AIN}(ain_{Am}(am))$  – it defines the *static* structure of a hierarchic state-machine. The dynamic structure is defined as internal state  $IntState_{Am}$  of abstract machines, and consists of a (1) *history* mapping and a (2) *current control point* mapping.

#### Internal State

The internal state of HybridUML abstract machines is defined by

$$IntState_{Am} = HIST \times CURCP$$

$$HIST = MIN_{spec} \rightarrow MIN_{spec} \cup \{\lambda\}$$

$$CURCP = MIN_{spec} \rightarrow CPIN \cup \{\lambda\}$$

The *history* mapping  $hist \in HIST$  assigns a currently active submode for each mode instance node, or  $\lambda$  to denote that there is none. For a given history and a root mode instance node, the history implies the current *mode configuration*

$$modeconf : HIST \times (MIN_{spec} \cup \{\lambda\}) \rightarrow \text{seq } MIN_{spec}$$

$$(hist, min) \mapsto \langle min \rangle \widehat{\text{modeconf}}(hist, hist(min)); min \neq \lambda$$

$$(hist, \lambda) \mapsto \langle \rangle$$

which is the sequence of active mode instance nodes descending the mode instance node tree. The mode configuration is *complete*, if the sequence ends with a leaf node, that is a node representing a mode that has no submodes. Note that the history further contains submodes of *inactive* mode instance nodes, to be restored for the mode configuration later.

The history always assigns mode instance nodes to their own submodes:

$$\begin{aligned} \forall hist \in HIST, min \in MIN_{spec} \bullet \\ hist(min) \neq \lambda \Rightarrow min = head(path_{MIN}(hist(min))) \end{aligned}$$

Each mode instance node may have a current control point instance node, defined by a mapping  $curcp \in CURCP$ , which defines a control point of the mode itself:

$$\begin{aligned} \forall curcp \in CURCP, min \in MIN_{spec} \bullet \\ curcp(min) \neq \lambda \Rightarrow min = min_{CPIN}(curcp(min)) \end{aligned}$$

The current control point mapping denotes which transitions are possibly enabled – only transitions that originate from a current control point may ever fire.

### Effect of Abstract Machines' Initialization

The initialization of abstract machines defines an internal state  $s_i \in IntState_{Am}$  such that (1) the history is empty, and (2) there is exactly one current control point – the default entry of the abstract machine's top-level mode instance node:

$$\begin{aligned} init_{Am} : Am \rightarrow IntState \\ am \mapsto (\{min \mapsto \lambda \mid min \in MIN_{spec}\}, \\ \{min \mapsto \lambda \mid min \in MIN_{spec}\} \oplus \{min \mapsto de_{min} \mid \\ min = node_{MIN_{spec}}(mtree_{AIN}(ain_{Am}(am))) \wedge \\ de_M(mode_{MI}(mi_{MIN}(min))) = cp_{CPI}(cpi_{CPIN}(de_{min}))\}) \end{aligned}$$

### Effect of Abstract Machines' Update

The updating of an abstract machine determines (1) its currently enabled transitions, (2) the enabledness of the abstract machine for a flow step, and (3) the enabling of particular flows for participation in such a flow step. This *update* functionality is defined step-by-step by use of several functions:

**Effect of Evaluation Programs.** For the determination of enabled transitions, flows, and flow-enabledness, the conditions that are given by HybridUML trigger expressions, guard expressions, and invariant constraint expressions, are evaluated. Therefore, programs that implement the evaluation interpretation of expression nodes are created and executed, and their boolean result is evaluated.

For this, we define the *effect of evaluation programs*. In contrast to the execution of HybridUML flow constraints and actions, the progress of evaluation programs for HybridUML triggers, guards, and invariant constraints wrt. time is *not* considered explicitly:

1. Assignment programs for HybridUML flow constraints and actions (and triggers) were defined in section 5.4.2, and are directly represented by HL<sup>3</sup> flows and transitions. They act as HL<sup>3</sup> program subjects, and their execution semantics is defined per program statement, by the progress rules for program statements in section 4.6.
2. Evaluation programs for HybridUML triggers, guards, and invariant constraints are defined below. For their execution, the effect of evaluation programs is defined on the basis of the same program statement effects from section 4.6, but no progress of time is defined here. However, for the complete *update* operation, an execution time duration is assumed by the progress rule for operations of abstract subjects (section 4.5.1).  
The programs that are generated for triggers, guards, and invariant constraints may be executed in parallel, within the HL<sup>3</sup> scheduling phase *update\_phase*, but they do not interfere, because they all use their own local variables (defined in section 5.4.1), and do not publish any values on channels. Therefore it is sufficient to consider the overall execution time for the *update* operation.

The effect of an execution of a complete evaluation program is given by

$$\begin{aligned}
\epsilon* &: Program \times Param_{prog} \times Subject \times CONST_m \times VAR_{mread} \\
&\rightarrow VAR_{mwrite} \\
(prg, param, s, c, v) &\mapsto \epsilon*(\epsilon_{prg}(prg, param, s, c, v), param, s, c, \\
&(\epsilon_{var}(prg, param, s, c, v), \epsilon_{chan}(prg, param, s, c, v))) \\
&; prg \neq \langle \rangle \\
(\langle \rangle, param, s, c, v) &\mapsto (\epsilon_{var}(prg, param, s, c, v), \epsilon_{chan}(prg, param, s, c, v))
\end{aligned}$$

For a program *prg*, the sequence of effects of the program's statements result in the effect of the program, as soon as the program is processed completely. The boolean result of the program is contained in the resulting variable valuation.

**Evaluation of Boolean Expression Nodes.** Each expression node that contains a boolean expression, as defined by HybridUML trigger expressions, guard expressions, and invariant constraint expressions, (1) defines an evaluation program, and (2) provides a boolean result on the execution of this program.

The program's definition is given by the mapping  $prg_{ExpN, eval}$  of section 5.5. Then, the boolean result of the execution of a program is given by its effect (defined by  $\epsilon*$ ), and the dedicated return variable's value is returned:

$$\begin{aligned}
exec_{bexp} &: BExpN_{spec} \times CONST_m \times VAR_{mread} \rightarrow \mathbb{B} \\
(expn, c, v) &\mapsto \\
&\sigma_{Var}(\epsilon*(prg_{ExpN, eval}(\langle expn \rangle), \lambda, ain_{ExpN}(expn), c, v)) \\
&(retvar_{BExpN}(expn))
\end{aligned}$$

**Filtering of Initial Transitions.** For the determination of enabled transitions, the first step is the filtering of initial transitions: Transitions that *initialize* modes can only be enabled, if the mode's history is empty. The following

mapping returns *false* for initial transition nodes of mode instance nodes with non-empty history:

$$\begin{aligned}
& \text{checkInitTrans}_{TN} : TN_{spec} \times \text{IntState}_{Am} \rightarrow \mathbb{B} \\
& (tn, (hist, curcp)) \mapsto \\
& \quad (\text{min}_{CPIN}(\text{src}_{TN_{spec}}(tn)) = \text{min}_{TN}(tn) \\
& \quad \wedge \text{cp}_{CPI}(\text{cpi}_{CPIN}(\text{src}_{TN_{spec}}(tn))) \\
& \quad \quad = \text{de}_M(\text{mode}_{MI}(\text{mi}_{MIN}(\text{min}_{TN}(tn)))))) \\
& \Rightarrow \text{hist}(\text{min}_{TN}(tn)) = \lambda
\end{aligned}$$

Here, always if (1) the transition's parent mode coincides with the parent mode of its source control point, (2) such that the source control point is the default entry point of that mode, then the mode's history must be empty for the given transition to be enabled.

**Enabling of Transitions.** Transition nodes can be enabled or disabled, depending on the trigger and guard expressions. For HybridUML, enabled transitions are *urgent*, if they have a trigger expression that is satisfied. Enabled transitions without trigger expression are not urgent, i.e. they do not need to be executed, but can. Additionally, initial transitions are disabled, whenever the parent mode has non-empty history.

$$\begin{aligned}
& \text{isEnabled}_{TN} : TN_{spec} \times \text{IntState}_{Am} \times \text{CONST}_m \times \text{VAR}_{mread} \rightarrow \mathbb{B} \\
& (tn, s_i, c, v) \mapsto \\
& \quad \text{checkInitTrans}_{TN}(tn, s_i) \wedge \\
& \quad \bigwedge_{txn \in \text{trg}_{TN_{spec}}(tn) \cup \text{grd}_{TN_{spec}}(tn)} \text{exec}_{beexp}(txn, c, v) \\
& \text{isUrgent}_{TN} : TN_{spec} \times \text{IntState}_{Am} \times \text{CONST}_m \times \text{VAR}_{mread} \rightarrow \mathbb{B} \\
& (tn, s_i, c, v) \mapsto \text{isEnabled}_{TN}(tn, s_i, c, v) \wedge \text{trg}_{TN_{spec}}(tn) \neq \emptyset
\end{aligned}$$

Note that trigger and guard expressions are evaluated by the execution of the corresponding evaluation programs, as defined before.

**Enabled Transitions of Control Points.** For each control point instance node, the set of enabled transitions consists of its outgoing transitions that are enabled:

$$\begin{aligned}
& \text{enabledTrans}_{CPIN} : CPIN_{spec} \times \text{IntState}_{Am} \times \text{CONST}_m \times \text{VAR}_{mread} \\
& \quad \rightarrow \mathcal{P}(TN_{spec}) \\
& (cpin, s_i, c, v) \mapsto \\
& \quad \{tn \in TN_{spec} \mid \text{src}_{TN_{spec}}(tn) = \text{cpin} \wedge \text{isEnabled}_{TN}(tn, s_i, c, v)\}
\end{aligned}$$

Analogously, urgent transitions are collected for control points:

$$\begin{aligned}
& \text{urgentTrans}_{CPIN} : CPIN_{spec} \times \text{IntState}_{Am} \times \text{CONST}_m \times \text{VAR}_{mread} \\
& \quad \rightarrow \mathcal{P}(TN_{spec}) \\
& (cpin, s_i, c, v) \mapsto \\
& \quad \{tn \in TN_{spec} \mid \text{src}_{TN_{spec}}(tn) = \text{cpin} \wedge \text{isUrgent}_{TN}(tn, s_i, c, v)\}
\end{aligned}$$

**Enabled Transitions of Mode Configurations.** For a mode configuration with root mode instance node  $min$ , the currently enabled transitions are collected: (1) The enabled transitions of the root mode's current control point are included, if there is one, and (2) all enabled transitions of its active submode are added recursively, if the history is not empty:

$$\begin{aligned}
& enabledTrans_{MIN} : MIN_{spec} \times IntState_{Am} \times CONST_m \times VAR_{mread} \\
& \quad \rightarrow \mathcal{P}(TN_{spec}) \\
& (min, (hist, curcp), c, v) \mapsto \{tn \in TN_{spec} \mid \\
& \quad \exists cpin \in CPIN_{spec} \bullet (curcp(min) = cpin \\
& \quad \wedge tn \in enabledTrans_{CPIN}(cpin, (hist, curcp), c, v)) \\
& \quad \vee \exists min_2 \in MIN_{spec} \bullet (hist(min) = min_2 \\
& \quad \wedge tn \in enabledTrans_{MIN}(min_2, (hist, curcp), c, v))\}
\end{aligned}$$

Similarly, urgent transitions are determined:

$$\begin{aligned}
& urgentTrans_{MIN} : MIN_{spec} \times IntState_{Am} \times CONST_m \times VAR_{mread} \\
& \quad \rightarrow \mathcal{P}(TN_{spec}) \\
& (min, (hist, curcp), c, v) \mapsto \{tn \in TN_{spec} \mid \\
& \quad \exists cpin \in CPIN_{spec} \bullet (curcp(min) = cpin \\
& \quad \wedge tn \in urgentTrans_{CPIN}(cpin, (hist, curcp), c, v)) \\
& \quad \vee \exists min_2 \in MIN_{spec} \bullet (hist(min) = min_2 \\
& \quad \wedge tn \in urgentTrans_{MIN}(min_2, (hist, curcp), c, v))\}
\end{aligned}$$

**Enabled Transitions of Abstract Machines.** The first part of the *update* result – the set of enabled transitions of an abstract machine – is given by the transition nodes that are enabled for the top-level mode of the abstract machine:

$$\begin{aligned}
& enabledTrans_{Am} : Am \times IntState_{Am} \times CONST_m \times VAR_{mread} \rightarrow \mathcal{P}(Trans) \\
& (am, s_i, c, v) \mapsto \{t \in Trans \mid tn_{trans}(t) \in \\
& \quad enabledTrans_{MIN}(node_{MIN_{spec}}(mtree_{AIN}(ain_{Am}(am))), s_i, c, v)\}
\end{aligned}$$

**Satisfied Invariant Constraints of Mode Configurations.** The first step for the determination of the second part of the *update* result, that is the check for the abstract machine's enabledness for a flow step, is the evaluation of the invariant constraints of mode configurations. Therefore, their conjunction is evaluated by

$$\begin{aligned}
& checkInv_{MIN} : MIN_{spec} \times IntState_{Am} \times CONST_m \times VAR_{mread} \rightarrow \mathbb{B} \\
& (min, (hist, curcp), c, v) \mapsto \bigwedge_{m \in inv_{MIN_{spec}}(min)} exec_{bexp}(m \backslash n, c, v) \\
& \quad \wedge ((\exists min_2 \in MIN_{spec} \bullet hist(min) = min_2) \\
& \quad \Rightarrow checkInv_{MIN}(min_2, (hist, curcp), c, v))
\end{aligned}$$

That is, the invariant constraints of the root mode are checked, and recursively the invariant constraints for the currently active submode, if there is one. Note that for the evaluation of invariant constraints, a corresponding program is executed, as defined before.

**Stable Mode Configurations.** Additionally, flow-enabledness requires a *stable* mode configuration. A mode configuration is stable, if it is complete, and for all active modes, the current control point is the respective *default exit point*. Therefore, for the mode configuration's root mode instance node, (1) the current control point is checked, and (2) if it is no leaf mode, then a non-empty history is required. (3) Finally, the history mode is examined recursively.

$$\begin{aligned}
& \text{checkDx}_{MIN} : MIN_{spec} \times \text{IntState}_{Am} \rightarrow \mathbb{B} \\
& ((s_M, mi, ain), (hist, curcp)) \mapsto \\
& \quad \exists (min_2, cpi) \in CPIN_{spec} \bullet (curcp(s_M, mi, ain) = (min_2, cpi) \\
& \quad \wedge cp_{CPI}(cpi) = dx_M(mode_{MI}(mi))) \\
& \quad \wedge \\
& \quad (submode_M(mode_{MI}(mi)) \neq \emptyset \\
& \quad \Rightarrow \exists min_2 \in MIN_{spec} \bullet \\
& \quad \quad min_2 = hist(min) \wedge \text{checkDx}_{MIN}(min_2, (hist, curcp)))
\end{aligned}$$

**Flow-Enabledness of Mode Configurations.** A mode configuration permits a flow of time, whenever (1) its invariants are satisfied, (2) the configuration is stable, and additionally, (3) there are no urgent transitions.

$$\begin{aligned}
& \text{isFlowPossible}_{MIN} : MIN_{spec} \times \text{IntState}_{Am} \times CONST_m \times VAR_{mread} \rightarrow \mathbb{B} \\
& (min, s_i, c, v) \mapsto \text{checkDx}_{MIN}(min, s_i) \wedge \text{checkInv}_{MIN}(min, s_i, c, v) \\
& \quad \wedge \text{urgentTrans}_{MIN}(min, s_i, c, v) = \emptyset
\end{aligned}$$

**Flow-Enabledness of Abstract Machines.** The permission of an abstract machine to let time pass – the second part of the *update* result – is determined by the top-level mode of the abstract machine:

$$\begin{aligned}
& \text{isFlowPossible}_{Am} : Am \times \text{IntState}_{Am} \times CONST_m \times VAR_{mread} \rightarrow \mathbb{B} \\
& (am, s_i, c, v) \mapsto \\
& \quad \text{isFlowPossible}_{MIN}(\text{node}_{MIN_{spec}}(\text{mtree}_{AIN}(\text{ain}_{Am}(am))), s_i, c, v)
\end{aligned}$$

**Activation of Associated Flow Expressions.** The third part of the *update* result is defined on the basis of a given mode instance node tree: All associated flow expressions are activated correspondingly to the tree's mode configuration: (1) On activation of the root mode's flow expression, the current active submode's flow expressions are activated, too, recursively. All inactive submodes' flow expressions are deactivated. (2) On deactivation of the root mode's flow expressions, all submodes' flow expressions are deactivated, too.

$$\begin{aligned}
& \text{activateFlows}_{MIN} : MIN_{spec} \times \text{IntState}_{Am} \times \mathbb{B} \rightarrow (\text{FlowExpN} \leftrightarrow \mathbb{B}) \\
& (min, (hist, curcp), b) \mapsto \\
& \quad \text{flow}_{MIN_{spec}}(min) \times \{b\} \cup \\
& \quad \bigcup_{min_2 \in \text{chld}} \text{activateFlows}_{MIN}(min_2, (hist, curcp), false) \cup \\
& \quad \bigcup_{min_2 \in \{hist(min)\} \setminus \{\lambda\}} \text{activateFlows}_{MIN}(min_2, (hist, curcp), b) \\
& \quad ; \text{chld} = \text{children}_{MIN_{spec}}(\text{tree}_{MIN}(min)) \setminus \{hist(min)\}
\end{aligned}$$

**Activation of Abstract Machines' Flow Expressions.** The flow expressions of an abstract machine's top-level mode are always *activated*. From the corresponding mode configuration, the activation or deactivation of flow expressions of submodes result.

$$\begin{aligned} & activateFlows_{Am} : Am \times IntState_{Am} \rightarrow (Flow \leftrightarrow \mathbb{B}) \\ & (am, s_i) \mapsto \{f \mapsto b \mid fn_{Flow}(f) \mapsto b \in \\ & \quad activateFlows_{MIN}(\text{node}_{MIN_{spec}}(\text{mtree}_{AIN}(\text{ain}_{Am}(am))), s_i, true)\} \end{aligned}$$

**Definition of *update*.** The effect of the operation *update* of a HybridUML abstract machine *am* is the combination of (1) the flow-enabledness of *am*, (2) the enabled transitions of *am*, and (3) the activation state of the associated flows:

$$\begin{aligned} & update : Am \times IntState \times CONST_m \times VAR_{mread} \\ & \quad \rightarrow IntState \times AmState \times (Flow \leftrightarrow \mathbb{B}) \\ & (am, s_i, c, v) \mapsto \\ & \quad (s_i, (isFlowPossible_{Am}(am, s_i, c, v), enabledTrans_{Am}(am, s_i, c, v)), \\ & \quad \quad activateFlows_{Am}(am, s_i)) \\ & \quad ; s_i \in IntState_{Am} \end{aligned}$$

Formally, the result of *update* for  $s_i \notin IntState_{Am}$  is arbitrary; but this is not effective since *init*<sub>Am</sub>, *update*, and *notify* always define a resulting state  $s_i \in IntState_{Am}$ .

Further note that HybridUML abstract machines only read, but do not modify their internal state on update.

### Effect of Abstract Machines' Notification

The notification of a HybridUML abstract machine occurs whenever in a preceding *transition\_phase* of the HL<sup>3</sup> model execution a transition was executed that is associated with the abstract machine. Then the abstract machine adjusts its internal state by taking the transition without executing its actions, such that the transition's source control point is left and its target control point is entered, potentially affecting the history as well.

**Entering of Control Points.** When a control point instance node *cpin* is entered, it becomes the current control point instance node of its parent mode instance node, i.e.  $curcp(\text{min}_{CPIN}(\text{cpin})) = \text{cpin}$ . The mapping

$$setcurcp : IntState_{Am} \times CPIN_{spec} \rightarrow CURCP$$

defines this, but the entering of a control point has several further implications. (1) If the control point is the default entry, then the history is resumed recursively for the mode, if possible:

$$\begin{aligned} & ((hist, curcp), \text{cpin}) \mapsto \\ & \quad setcurcp((hist, curcp \oplus \{\text{min}_{CPIN}(\text{cpin}) \mapsto \text{cpin}\}), de_h) \\ & \quad ; cp_{CPI}(\text{cp}_{CPI}(\text{cpin})) = de_M(\text{mode}_{MI}(\text{mi}_{MIN}(\text{min}_{CPIN}(\text{cpin})))) \\ & \quad \wedge \text{min}_{CPIN}(de_h) = hist(\text{min}_{CPIN}(\text{cpin})) \\ & \quad \wedge cp_{CPI}(\text{cp}_{CPI}(de_h)) = de_M(\text{mode}_{MI}(\text{mi}_{MIN}(\text{min}_{CPIN}(de_h)))) \end{aligned}$$

(2) If for a leaf mode the control point is the default entry, then it is directly transferred to the default exit:

$$\begin{aligned} & ((hist, curcp), cpin) \mapsto setcurcp((hist, curcp), dx) \\ & ; cp_{CPI}(cpi_{CPIN}(cpin)) = de_M(mode_{MI}(mi_{MIN}(min_{CPIN}(cpin)))) \\ & \quad \wedge cp_{CPI}(cpi_{CPIN}(dx)) = dx_M(mode_{MI}(mi_{MIN}(min_{CPIN}(cpin)))) \\ & \quad \wedge submode_M(mode_{MI}(mi_{MIN}(min_{CPIN}(cpin)))) = \emptyset \end{aligned}$$

(3) If the control point is the default exit, then all parent modes are also set to their default exits recursively:

$$\begin{aligned} & ((hist, curcp), cpin) \mapsto \\ & \quad setcurcp((hist, curcp \oplus \{min_{CPIN}(cpin) \mapsto cpin\}), dx_p) \\ & ; cp_{CPI}(cpi_{CPIN}(cpin)) = dx_M(mode_{MI}(mi_{MIN}(min_{CPIN}(cpin)))) \\ & \quad \wedge cp_{CPI}(cpi_{CPIN}(de_p)) = dx_M(mode_{MI}(mi_{MIN}(min_{CPIN}(de_p)))) \\ & \quad \wedge mi_{MIN}(min_{CPIN}(cpin)) \\ & \quad \in submode_M(mode_{MI}(mi_{MIN}(min_{CPIN}(dx_p)))) \end{aligned}$$

(4) By default, the current control point of the mode is just set:

$$((hist, curcp), cpin) \mapsto curcp \oplus \{min_{CPIN}(cpin) \mapsto cpin\}; \text{ else}$$

**Leaving of Control Points.** Leaving of a control point instance node  $cpin$  assigns the special value  $\lambda$  to the parent mode instance node, in order to indicate that it has no current control point (anymore):  $curcp(min_{CPIN}(cpin)) = \lambda$ . This is defined by the mapping

$$\begin{aligned} & unsetcurcp : IntState_{Am} \times CPIN_{spec} \rightarrow CURCP \\ & ((hist, curcp), cpin) \mapsto unsethistcp((hist, unsetparcp(curcp \oplus \\ & \quad \{min_{CPIN}(cpin) \mapsto \lambda\}, min_{CPIN}(cpin))), min_{CPIN}(cpin)) \end{aligned}$$

The complete mode configuration gets unset automatically, i.e. the control points of all parent modes and history modes are unset, too, recursively in both cases:

$$\begin{aligned} & unsetparcp : CURCP \times MIN_{spec} \rightarrow CURCP \\ & (curcp, min) \mapsto unsetparcp(curcp \oplus \{min_p \mapsto \lambda\}, min_p) \\ & \quad ; mi_{MIN}(min) \in submode_M(mode_{MI}(mi_{MIN}(min_p))) \\ & (curcp, min) \mapsto curcp; \text{ else} \end{aligned}$$

$$\begin{aligned} & unsethistcp : IntState_{Am} \times MIN_{spec} \rightarrow CURCP \\ & ((hist, curcp), min) \mapsto unsethistcp((hist, curcp \oplus \{min_h \mapsto \lambda\}), min_h) \\ & \quad ; min_h \neq \lambda \wedge min_h = hist(min) \\ & ((hist, curcp), min) \mapsto curcp; \text{ else} \end{aligned}$$

**Modification of the History.** On the firing of a transition, the history is modified, corresponding to the transition's target control point:

$$newhist : HIST \times TN_{spec} \rightarrow HIST$$

In the majority of cases, transitions (indirectly) connect submodes, such that the history of their own parent mode is set to the mode of the target control point. Because of the syntactical constraints for HybridUML transitions (see section 2.2), this situation is identified by the direction of the control point, i.e. when it is an entry point:

$$\begin{aligned} (hist, tn) &\mapsto hist \oplus \{min_{TN}(tn) \mapsto min_{CPI}(tar_{TN_{spec}}(tn))\} \\ &; kind_{CP}(cp_{CPI}(tar_{TN_{spec}}(tn))) = entry \end{aligned}$$

Otherwise, i.e. if the control point is an exit point, the transition leads to the exit point of its own parent mode. Since this must not be the default exit (for the same syntactical restrictions as above), the history is *cleared* for that mode:

$$\begin{aligned} (hist, tn) &\mapsto hist \oplus \{min_{TN}(tn) \mapsto \lambda\} \\ &; kind_{CP}(cp_{CPI}(tar_{TN_{spec}}(tn))) = exit \end{aligned}$$

**Firing of Transitions.** A transition fires by (1) leaving its source control point, (2) modifying its parent mode's history, and (3) entering its target control point:

$$\begin{aligned} fire &: IntState_{Am} \times TN_{spec} \rightarrow IntState_{Am} \\ ((hist, curcp), tn) &\mapsto (setcurcp( \\ &\quad (newhist(hist, src_{TN_{spec}}(tn)), \\ &\quad unsetcurcp((hist, curcp), src_{TN_{spec}}(tn))), \\ &\quad tar_{TN_{spec}}(tn)), \\ &\quad newhist(hist, src_{TN_{spec}}(tn))) \end{aligned}$$

**Definition of *notify*.** The effect of the operation *notifyTrans* of a HybridUML abstract machine *am* is then defined by the firing of the given transition, provided that it is a transition of *am*:

$$\begin{aligned} notify &: Am \times IntState \times Trans \rightarrow IntState \\ (am, s_i, t) &\mapsto fire(s_i, tn_{trans}(t)); s_i \in IntState_{Am} \wedge am_{huml,trans}(t) = am \\ (am, s_i, t) &\mapsto s_i; \text{ else} \end{aligned}$$

## 5.6.2 HybridUML Selector

The selector defined in this section is the *HybridUML simulation selector*. It is tailored for the simulation of HybridUML specifications, such that a maximum set of HybridUML-specific HL<sup>3</sup> model executions is defined. Any non-determinism of the HybridUML specification is actually simulated non-deterministically. In contrast, a *HybridUML test selector* would restrict the set of executions by applying some elaborate test selection algorithm, or a *HybridUML implementation selector* could solve non-determinism by making fixed choices.

### Internal State

The HybridUML selector only has internal state that models non-determinism. We do not define this explicitly, but assume that there is a function

$$rndstep_{selector} : IntState_{sel} \rightarrow IntState_{sel}$$

that calculates some kind of random sequence on the internal state.

### Effect of Selector's Initialization

The initialization of the selector has two separate effects: (1) Corresponding to the notion of the internal state of the selector given above, we assume that the initialization defines an appropriate starting point for the random sequence modeling non-determinism. (2) Since the selector has the responsibility to check the initial HL<sup>3</sup> model state for well-formedness, the initial valuation of the model's channels is checked whether it satisfies the init state constraints of the agent instance nodes, represented by the abstract machines of the HL<sup>3</sup> model.

**Initial State Well-Formedness Check.** In order to check the init state constraints of the HybridUML model (see also sections 5.4 and 3.1 for discussions of init state constraints), expression nodes that represent the constraints are evaluated. This is done in the same fashion as for trigger expressions, guard expressions, and invariant constraint expressions in section 5.6.1, for the definition of the abstract machine's operation *update*. Expression nodes define a respective program that is executed and that returns a boolean result value. The initial state is well-formed, if the conjunction of all constraints' results is satisfied:

$$\begin{aligned} check_{initState} : S \rightarrow \mathbb{B} \\ (c, v) \mapsto \bigwedge_{ixn \in isc_{Am}(c, v)} \quad & exec_{bexp}(ixn, \\ & (subject_{var}(c), chan_{port}(c), subject_{port}(c)), \\ & (modelTime(v), \sigma_{Var}(v), \kappa_{Chan}(v))) \end{aligned}$$

The init state constraints are provided by the abstract machines of the HL<sup>3</sup> model:

$$\begin{aligned} isc_{Am} : S \rightarrow \mathcal{P}(BExpN_{spec}) \\ (c, v) \mapsto \bigcup_{m \in am(c)} allisc_{AIN_{spec}}(ain_{Am}(m)) \end{aligned}$$

**Definition of  $init_{sel}$ .** The effect of the operation *init* of the HybridUML simulation selector is (1) the creation of the internal state, and (2) the well-formedness check of the channels' initial valuation:

$$\begin{aligned} init_{sel} : S \rightarrow IntState \times \mathbb{B} \\ s \mapsto (s_i, check_{initState}(s)) \\ \text{with appropriate internal state } s_i \end{aligned}$$

Remember from section 4.5.2 that the consequence of a *failed* well-formedness check is that there is no valid execution of the complete HL<sup>3</sup> model.

### Effect of Selector's Selection

The calculation of a selection by the selector either (1) defines a set of transitions to be executed in a *transition\_phase*, or (2) chooses a *flow\_phase*, such that the currently activated flows are calculated. Additionally, HybridUML signals are unset before a succeeding *flow\_phase*, in order to implement a zero-time duration for signals.

In the following, several functions are defined and combined to the complete functionality of the selector's operation *getSelection*.

**Flow-Enabledness of the Simulation.** The possibility of a flow step of the simulation is determined in a straight-forward way: a flow step is only admissible, if all abstract machines allow it:

$$\begin{aligned} \text{selectflow} &: \Sigma_{Am} \rightarrow \mathbb{B} \\ s_{Am} &\mapsto \forall m \in \text{dom } s_{Am} \bullet \text{flow}_{AmState}(s_{Am}(m)) \end{aligned}$$

**Collection of Enabled Transitions.** For the selection of transitions, the separate sets of transitions from the abstract machines are collected, such that a set of transition sets results that contains the set of enabled transitions for each abstract machine:

$$\begin{aligned} \text{collecttrans} &: \Sigma_{Am} \rightarrow \mathcal{P}(\mathcal{P}(\text{Trans})) \\ s_{Am} &\mapsto \{trs \in \mathcal{P}(\text{Trans}) \mid \exists m \in \text{dom } s_{Am} \bullet trs = \text{trans}_{AmState}(s_{Am}(m))\} \end{aligned}$$

**Non-Conflicting Transitions.** The HybridUML semantics is an *interleaving semantics*, i.e. logically there are no parallel HL<sup>3</sup> transitions (i.e. discrete steps), but transitions are executed sequentially. The strict definition of this interleaving would lead to transition phases of the HL<sup>3</sup> model execution that only execute one single HL<sup>3</sup> transition, with the drawback that for  $n$  available light weight processes, only one could be active, while the others would be idle. As an optimization of the execution (with  $n > 1$ ), transitions that are *non-conflicting* can be executed in parallel, without modifying the interleaving semantics.

Since local HL<sup>3</sup> variables are exclusively accessed by each subject, and therefore no racing conditions occur *during* the parallel execution, only the execution *sequence* is significant. Two transitions are non-conflicting, whenever neither of both transitions relies on the results of the other one, such that any execution sequence leads to the same result, which is the result of the logical sequence of the interleaving semantics.

This is guaranteed, if the HL<sup>3</sup> channels from which one transition reads values (the *read set*) are disjoint from the channels on which the other transition writes (the *write set*):

$$\begin{aligned} \text{nonconfl} &: \text{Trans} \times \text{Trans} \rightarrow \mathbb{B} \\ (t_1, t_2) &\mapsto \text{readset}(t_1) \cap \text{writeset}(t_2) = \emptyset \\ &\quad \wedge \text{readset}(t_2) \cap \text{writeset}(t_1) = \emptyset \end{aligned}$$

For each two transitions, *nonconfl* maps to *true* if the transitions are non-conflicting, and to *false* if they are in conflict.

The read sets and write sets are defined as mappings from transitions to sets of channels. All channels for HybridUML variables and signals are collected

for which an expression node exists, such that the variable or signal is accessed (read or written, resp.) by the corresponding expression. From section 5.3, mappings to determine variables and signals that are accessed by expressions are applied:

$$\begin{aligned}
\text{readset} : \text{Trans} &\rightarrow \mathcal{P}(\text{CHN}_{\text{Var}} \cup \text{CHN}_{\text{Sig}}) \\
t &\mapsto \{c \in \text{CHN}_{\text{Var}} \mid \exists vn \in \text{vn}_{\text{conn,max,CHN}_{\text{Var}}}(c) \bullet \\
&\quad \exists \text{expn} \in \text{trg}_{\text{TN}_{\text{spec}}}(\text{tn}_{\text{trans}}(t)) \cup \text{ran } \text{act}_{\text{TN}_{\text{spec}}}(\text{tn}_{\text{trans}}(t)) \bullet \\
&\quad \text{var}_{\text{VN}}(vn) \in \text{var}_{\text{ht,read}}(\text{ht}_{\text{expn}}(\text{expn}))\} \\
&\cup \\
&\{c \in \text{CHN}_{\text{Sig}} \mid \exists sn \in \text{sn}_{\text{conn,max,CHN}_{\text{Sig}}}(c) \bullet \\
&\quad \exists \text{expn} \in \text{trg}_{\text{TN}_{\text{spec}}}(\text{tn}_{\text{trans}}(t)) \cup \text{ran } \text{act}_{\text{TN}_{\text{spec}}}(\text{tn}_{\text{trans}}(t)) \bullet \\
&\quad \text{sig}_{\text{SN}}(sn) \in \text{sig}_{\text{ht,read}}(\text{ht}_{\text{expn}}(\text{expn}))\} \\
\\
\text{writeset} : \text{Trans} &\rightarrow \mathcal{P}(\text{CHN}_{\text{Var}} \cup \text{CHN}_{\text{Sig}}) \\
t &\mapsto \{c \in \text{CHN}_{\text{Var}} \mid \exists vn \in \text{vn}_{\text{conn,max,CHN}_{\text{Var}}}(c) \bullet \\
&\quad \exists \text{expn} \in \text{trg}_{\text{TN}_{\text{spec}}}(\text{tn}_{\text{trans}}(t)) \cup \text{ran } \text{act}_{\text{TN}_{\text{spec}}}(\text{tn}_{\text{trans}}(t)) \bullet \\
&\quad \text{var}_{\text{VN}}(vn) \in \text{var}_{\text{ht,write}}(\text{ht}_{\text{expn}}(\text{expn}))\} \\
&\cup \\
&\{c \in \text{CHN}_{\text{Sig}} \mid \exists sn \in \text{sn}_{\text{conn,max,CHN}_{\text{Sig}}}(c) \bullet \\
&\quad \exists \text{expn} \in \text{trg}_{\text{TN}_{\text{spec}}}(\text{tn}_{\text{trans}}(t)) \cup \text{ran } \text{act}_{\text{TN}_{\text{spec}}}(\text{tn}_{\text{trans}}(t)) \bullet \\
&\quad \text{sig}_{\text{SN}}(sn) \in \text{sig}_{\text{ht,write}}(\text{ht}_{\text{expn}}(\text{expn})) \cup \text{sig}_{\text{ht,read}}(\text{ht}_{\text{expn}}(\text{expn}))\}
\end{aligned}$$

Note that read signals are included in the write set, because signals are consumed on reception, and therefore the corresponding value is written on the respective channel.

**Sets of Non-Conflicting Transitions.** From sets of transition sets, every possible allowed transition combination can be generated, such that (1) at most one transition per input set is included, and (2) there are no conflicting transitions in each output set. This prepares the selection of a set of transitions of the HL<sup>3</sup> model for a (possibly) succeeding *transition\_phase*. Every abstract machine can take one transition at most, because their behavior is *sequential*. Their transitions may not interfere, in order to guarantee a valid execution optimization for an arbitrary interleaving of the transitions, because every possible execution sequence of these transitions is allowed.

$$\begin{aligned}
\text{nonconfltrans} : \mathcal{P}(\mathcal{P}(\text{Trans})) &\rightarrow \mathcal{P}(\mathcal{P}(\text{Trans})) \\
\{trs_1, \dots, trs_n\} &\mapsto \{trs \in \mathcal{P}(\bigcup_{i=1}^n trs_i) \mid \forall t_1, t_2 \in trs, k, l \in \{1..n\} \bullet \\
&\quad ((t_1 \in trs_k \wedge t_2 \in trs_l \wedge t_1 \neq t_2 \Rightarrow trs_k \neq trs_l) \\
&\quad \wedge \text{nonconfl}(t_1, t_2))\}
\end{aligned}$$

**Choice of a Transition Set.** From the set of allowed transition combinations, one is chosen:

$$\text{selecttrans} : \Sigma_{Am} \times \text{IntState}_{\text{sel}} \rightarrow \mathcal{P}(\text{Trans})$$

$$(s_{Am}, s_i) \mapsto trs$$

such that

$$trs \in nonconfltrans(collecttrans(s_{Am}))$$

$$\wedge (\exists trs_2 \in nonconfltrans(collecttrans(s_{Am})) \bullet trs_2 \neq \emptyset) \Rightarrow trs \neq \emptyset$$

Here, mainly the given internal selector state determines which of the available transition sets is chosen. The choice is not entirely non-deterministic, because  $trs \neq \emptyset$  is always preferred. Otherwise the HL<sup>3</sup> model execution could step through any number of empty transition phases, and therefore the model's real-time execution could fail (see section 4.4.1 for successful model executions) even if successful runs exist for the model.

**Choice of Flow or Transition Phase.** Based on the availability of transitions and the enabledness for a flow step, the selector chooses either a *transition\_phase* or a *flow\_phase*. (1) If both transitions are available and a flow step is possible, depending on the internal selector state, a random choice is made, such that either the transitions are removed or the flow step is disabled. (2) Otherwise, the transition set and the flow flag are left untouched. HybridUML models exist for which a *transition\_phase* with empty transition set can result here, they are deemed to be *not well-formed*.

$$selectflowortrans : \Sigma_{Am} \times IntState_{sel} \rightarrow \mathbb{B} \times \mathcal{P}(Trans)$$

$$(s_{Am}, s_i) \mapsto (flow, trs)$$

$$(selectflow(s_{Am}) \wedge selecttrans(s_{Am}, s_i) \neq \emptyset) \Rightarrow$$

$$((flow, trs) \in \{(true, \emptyset), (false, selecttrans(s_{Am}, s_i))\})$$

$$(\neg selectflow(s_{Am}) \vee selecttrans(s_{Am}, s_i) = \emptyset) \Rightarrow$$

$$((flow, trs) = (selectflow(s_{Am}), selecttrans(s_{Am}, s_i)))$$

**Creation of a Selection.** To all transitions which are selected, a visibility set is added that defines when the results of the respective action would become visible. For HybridUML, this is always the current tick, incremented by 0.1, i.e. no time passes, but causality between succeeding transition phases is modeled. This publication time tick is the same for all potential recipients:

$$selection : \Sigma_{Am} \times IntState_{sel} \times ModelTime \rightarrow Selection$$

$$(s_{Am}, s_i, tick) \mapsto (\pi_1 selectflowortrans(s_{Am}, s_i),$$

$$\{tr \mapsto Port \times \{tick + 0.1\} \mid tr \in \pi_2 selectflowortrans(s_{Am}, s_i)\})$$

**Resetting of Signals.** As a side-effect of the selection, the HybridUML selector resets signals whenever model time evolves, i.e. when it selects a flow phase. This ensures that signals have no time duration. To all available signal channels, the value *false* is written for all possible recipients, in order to indicate that no signal is active anymore. For this, it is assumed that the given port set  $P$  contains ports for exactly the respective channels. The publication time is

the current time tick, increased by 0.1, which will be earlier than current model time in the next update phase (when signals will be read next). That means, signals are reset immediately.

$$\begin{aligned} \text{resetSignals} &: \text{ModelTime} \times \mathcal{P}(\text{Port}) \rightarrow \text{ChanState} \\ (tick, P) &\mapsto (\{tick + 0.1\} \times P) \times \{false\} \end{aligned}$$

**Definition of *select*.** The main effect of the operation *getSelection* of the HybridUML simulation selector is the selection between *flow\_phase* and *transition\_phase*, along with the transitions and their visibility sets. Additionally, the signal channels' state may be modified, due to the resetting of signals. Finally, the internal state is adjusted.

$$\begin{aligned} \text{select} &: \text{ModelTime} \times \mathcal{P}(\text{Port}) \times \text{IntState} \times \Sigma_{Am} \\ &\rightarrow \text{IntState} \times \text{Selection} \times \text{ChanState} \\ (tick, P, s_i, s_{Am}) &\mapsto (\text{rndstep}_{\text{selector}}(s_i), \text{selection}(s_{Am}, s_i, tick), s_{chan}) \\ \text{with } s_{chan} &= \begin{cases} \text{resetSignals}(tick, P) & \text{if } \pi_1 \text{selectflowortrans}(s_{Am}, s_i) \\ \emptyset & \text{else} \end{cases} \end{aligned}$$



# Chapter 6

## Conclusion

This chapter concludes this thesis. A summary is given, and the main scientific contributions are pointed out. Possible future work related to HybridUML, to the low-level framework HL<sup>3</sup>, and to the transformational approach is discussed in short.

### 6.1 Summary

We have motivated the use of the specification formalism HybridUML for the formal definition of hybrid systems. The topic of hybrid systems and hybrid systems modeling has been introduced, and the need for usable formal specification facilities has been pointed out, with an emphasis on the importance of automatic generation of an executable system.

As an approach to combine the definition of hybrid systems in a formal, but user-friendly way, with the generation of a resulting executable system that has formally defined behavior, we have proposed a transformation concept: Hybrid systems models are modeled with the specification language HybridUML, and are transformed into programs of the Hybrid Low-Level Framework HL<sup>3</sup>, which provides a restricted design pattern that the transformation has to comply with, as well as a runtime environment that provides basic functionality. The semantics of the low-level framework HL<sup>3</sup>, as well as the transformation of HybridUML models into HL<sup>3</sup> are defined formally, therefore the semantics of HybridUML models are given by the semantics of the corresponding HL<sup>3</sup> programs. Therefore, this approach is superior to the separate definition of a high-level semantics and implementation of an executable system, in that there is no gap between the semantics and the actual behavior of the implementation.

We have presented the specification formalism HybridUML, which combines the usability of the Unified Modeling Language and the expressiveness given by the executable semantics defined by the transformation to HL<sup>3</sup>. We have sketched HybridUML's graphical notation along with the intuition of its execution. This has been done by means of a case study about the specification of a radio-based train control system. Instead of using the graphical HybridUML syntax directly, we have defined a data structure that exactly represents HybridUML models syntactically, but non-graphically. In terms of UML, this data structure is the HybridUML meta-model. This separation of the meta-model from its graphical representation is the usual UML approach. The benefits are

that the meta-model is directly usable for the transformation  $\Phi$ , and that the HybridUML semantics is independent from the graphical notation.

The syntax of the HybridUML Expression Language has been defined separately from the syntax of HybridUML, and its intuitive semantics has been explained informally. The HybridUML Expression Language defines the expressions that can be used within a HybridUML model, e.g. boolean expressions from mode invariants or transition conditions, or assignment expressions from transition actions. We have given an intermediate semantics, which is a data structure needed for the definition of its formal semantics. The semantics has been given in the context of the transformation  $\Phi$ , by the definition of transformation rules that map expressions to programs.

We have defined the HL<sup>3</sup> Low-Level Framework, a compilation target for hybrid systems specification formalisms. HL<sup>3</sup> restricts the variety of executable programs to a fixed structure – the HL<sup>3</sup> design pattern – in a way that is supposed to be adequate for arbitrary formalisms that provide the modeling of hybrid systems. Any transformation  $\Phi$  from such a specification formalism into HL<sup>3</sup> must preserve these restrictions. Further, HL<sup>3</sup> provides fixed components that can be used by transformations  $\Phi$  to construct HL<sup>3</sup> models from corresponding high-level models. The available components are called the HL<sup>3</sup> runtime environment.

A formal operational semantics has been given for the execution of HL<sup>3</sup> models. Therefore, HL<sup>3</sup> models are defined as a mixture of explicit program code and abstractions to mathematical representations. The specific program code of a specific model, and parts of the abstractly defined behavior, depend on the particular specification formalism, as well as on the specific high-level model to be transformed.

Finally, the specific transformation  $\Phi_{HUML}$  from HybridUML models to instances of the HL<sup>3</sup> design pattern has been defined formally. The resulting semantics is the HybridUML simulation semantics, which defines the behavior of self-contained simulations of complete HybridUML models. The transformation results are separated into two parts: (1) Independently from the specific model, HybridUML-specific behavior definitions are given. (2) Corresponding to the specific HybridUML model, the entities of HL<sup>3</sup> model, as well as their dependencies, are defined.

## 6.2 Contributions

The main contributions of this thesis are based on the transformation concept from hybrid systems specification formalisms to executable models of a low-level language, which has been joint work of the authors from [BBHP04a]: (1) From the semi-formal definition of HybridUML in [BBHP03], we have constructed a mathematically formal HybridUML syntax. In addition, we have defined the HybridUML Expression Language formally. (2) Based on the initially informal proposal in [BBHP04a], we have constructed a full definition of the HL<sup>3</sup> Low-Level Framework, and have provided a formal operational semantics for it. (3) We have defined the specific transformation  $\Phi_{HUML}$  that maps HybridUML models to executable HL<sup>3</sup> models formally, and therefore have defined an operational semantics for HybridUML models, as well as an executable simulation thereof.

For the proof of concept of the transformation  $\Phi_{HUMML}$ , we have implemented the HybridUML Mathematical Meta-Model, the HybridUML Expression Language the transformation  $\Phi_{HUMML}$  itself, and a subset of the HL<sup>3</sup> Low-Level Framework: (1) The HybridUML Mathematical Meta-Model is implemented as a C++ class library that can be used to instantiate HybridUML model elements syntactically. For example, the export of a graphical HybridUML model from a UML CASE tool can be realized by linking the library and creating the corresponding set of objects. (2) For an ASCII-based variant of HyBEL expressions, a corresponding scanner and parser were developed (with the generator tools *flex* and *bison*). (3) The HL<sup>3</sup> framework is implemented as object-oriented C++ library, consisting of abstract classes corresponding to the design pattern, as well as classes implementing the runtime environment. The main limitations of the implementation are that it is *not* optimized for speed, and that only a single light weight process, i.e. no real parallelism is supported so far. (4) The model-independent part of the transformation  $\Phi_{HUMML}$  is implemented as specialized class definitions, which add the HybridUML-specific behavior to the respective HL<sup>3</sup> entities. (5) The model-dependent part of the transformation  $\Phi_{HUMML}$  is implemented as C++ program which links the specific HybridUML model instance and generates the corresponding C++ code that completes the specialized framework, resulting in a C++ program that can be compiled and executed.

As an application example, a major subset of the case study *Radio-Based Train Control* has been realized. The complete behavior of the train, as well as the complete behavior of the crossing have been defined. We have created two consistent versions – a graphical one, corresponding to the illustrated graphical HybridUML notation of section 1.3, and an instance of the implemented HybridUML Mathematical Meta-Model. The first is given in appendix C, and the latter has been used to test the implemented transformation  $\Phi_{HUMML}$ , as well as the implemented HL<sup>3</sup> framework. Excerpts of the C++ variant of the resulting HL<sup>3</sup> program are presented in appendix D.

## 6.3 Future Work

We partition our vision of future work in two sections: On the one hand, from restrictions of the work in this thesis, highly desirable *enhancements* result that would add major benefits to the presented transformation approach, and are briefly discussed in section 6.3.1. On the other hand, *further investigation* topics that are beyond of the presented approach are sketched in section 6.3.2.

### 6.3.1 Enhancements

**Physical and Architectural Specification.** A major limitation of HybridUML and the transformation  $\Phi_{HUMML}$  is, that there are no *Architectural Specification* and no *Physical Constraints Specification*. Particularly the Architectural Specification is needed to distinguish different *roles* of structural entities of HybridUML specifications (i.e. agents). Roles could define whether an agent is *internal* or *external* to the simulation, and therefore facilitate the integration of external (hardware) components into the execution of the corresponding HL<sup>3</sup> model.

In terms of the transformation to HL<sup>3</sup>, external agents would be mapped to *interface modules*, rather than to *abstract machines*, as it is done so far for all agents of a HybridUML model. Only self-contained simulations of complete HybridUML models are defined, currently.

Further aspects, like the distribution of agents to light weight processes, or the definition of discretization frequencies for continuous calculations, would be useful for optimization purposes.

**Implementation Optimization.** As a direct consequence of the implementation limitations noted above, the HL<sup>3</sup> implementation (1) should be revised and optimized for speed, and (2) support for more than one light weight process should be added. A promising approach for this is to merge our implementation with the implementation of the RT-Tester tool [VS04]. This tool contains a related runtime environment to that of HL<sup>3</sup>, but is tailored to the execution of discrete specifications. For example, it is used for embedded systems testing of controllers for the Airbus A380 aircraft family. Test engines operate with cluster configurations consisting of 3, 5, or more multi-CPU PC nodes.

**Constraint Solving for HL<sup>3</sup> Initial State.** A slightly less critical, but nonetheless convenient enhancement would be the integration of a constraint solver that realizes the determination of a *set of* initial HL<sup>3</sup> states, on the basis of HybridUML Init State Constraints (see 5.4.2). Especially for hybrid systems models which include environmental components, it is often not feasible to ensure that a single dedicated initial state holds when the computer system within the system shall be activated. Particularly, the initial valuation of continuous environmental variables will always be in a range of values, rather than be exactly defined. In the presence of an integrated constraint solver, a set of HL<sup>3</sup> models could be derived in a convenient way.

**Validation of HL<sup>3</sup> and  $\Phi_{HUML}$ .** In this thesis, we have *constructed* the HL<sup>3</sup> framework, as well as the transformation  $\Phi_{HUML}$ . We have not proven formally, that (1) HL<sup>3</sup> models are well-formed wrt. “conventional” properties, e.g. that the operational semantics itself is dead-lock free, or that executions of HL<sup>3</sup> models are non-zeno. Further, (2) we have not validated the transformation  $\Phi_{HUML}$ , for example, that the HybridUML selector does not introduce dead-locks, or that the transformation algorithm terminates for arbitrary HybridUML models.

Both would be helpful to confirm the transformational approach of this thesis. Actually, the first is currently under preparation.

### 6.3.2 Further Investigations

**Testing and Test Data Generation.** As automated testing and test data generation is a main topic of research for the *Research Group Operating Systems, Distributed Systems* of the University of Bremen, of course the benefits of HybridUML, HL<sup>3</sup>, and the transformation  $\Phi_{HUML}$  for the purpose of testing are to be investigated. See [Pel96] for the formal background of testing.

An important difference to the simulation semantics of HybridUML is that components of the complete hybrid system play different roles, in the sense discussed above (section 6.3.1). Additionally, not only a distinction is made

between internal and external components, but internal components contribute to (1) the simulation of the *operational environment*, to (2) the *test data generation*, or to (3) the *test data evaluation*, whereas the *System Under Test (SUT)* is given as external component. The simplest testing approach is the simulation execution with the external SUT, provided that an Architectural Specification facility exists (see section 6.3.1).

For a more elaborate generation of test data, with regard to a required *test coverage*, the transformation  $\Phi_{HUMML}$  could be modified, resulting in a transformation  $\Phi_{HUMML, test}$ . From HybridUML specifications of the SUT's intended behavior, or probably from the specification of the operational environment, specialized HL<sup>3</sup> abstract machines can probably be constructed, along with a specialized *HybridUML test selector*.

The definition of  $\Phi_{HUMML, test}$  could aim to exploit the structure of the HybridUML model. On the basis of the structure of flat automata, the classical W-method [Cho78] is an algorithm to generate a finite set of input sequences for the SUT which is (almost) equivalent to testing all possible input sequences, and therefore provides some kind of “full” coverage. [Bin00] adapts this approach to discrete hierarchic statecharts.

For HybridUML statecharts, the main contribution is to take the continuous behavioral aspects into account.

**Formal Reasoning.** For the purpose of formal reasoning about properties of a hybrid system, two different approaches are conceivable: (1) A HybridUML high-level semantics, i.e. a more abstract semantics could complement the transformational approach presented in this thesis. The main proof obligation for the existence of two separate semantics definitions is then to show that they are related in a reasonable way. For the assumption that the discretization of the transformational semantics operates infinitely fast, one would expect some kind of refinement relation between them. A corresponding semantics for HybridUML is currently under construction, that does not define a discretization of continuous evolutions, and does not model light weight processes, for example. (2) Alternatively, high-level models could be augmented by property specifications that are preserved by the transformation into the low-level model, such that they would be checked on the basis of the HL<sup>3</sup> framework only. It has to be investigated which additions to high-level models and to transformations  $\Phi$  are therefore necessary. The expected benefit is that formal reasoning could be accomplished independently of the specific high-level formalism, to a certain degree.

**Generic HybridUML and Domain-Specific Specializations.** Targeting the reusability of HybridUML, the extension of HybridUML by generic concepts is sketched in [OH05] (but not defined formally), and illustrated by an example from the avionics domain. By the application of syntactical refinement and substitution of model elements, a hierarchy of HybridUML models is proposed that contains a very abstract aircraft model at the top. In several steps, more specialized aircraft family models, and finally specifically customized aircraft models are refined from this. In this fashion, domain specific knowledge could be structured and reused systematically.

A further step on top of HybridUML would be to adapt the HybridUML

syntax and graphical notation for specific domains, encoding domain-specific knowledge into the language itself. For this, again the profiling mechanism of UML could be applied, like for the initial definition of HybridUML [BBHP03]. For investigations on domain-specific languages apart from HybridUML see [HP03, HP02], for example.

**Virtual Reality Abstractions.** In order to focus on the aspect of user-acceptance and -friendliness, in combination with domain-specific specializations of HybridUML, a topic of research is the applicability of domain-specific virtual reality representations in order to generate (parts of) formal HybridUML specifications. The goal would be to reduce the need for formal methods expertise and to concentrate on the domain-specific knowledge of domain experts. Some efforts have already been taken on this topic, with a focus on the creation of test specifications [BT02b, BT02a, PBF99, BF99].



Figure 6.1: Train's Cockpit, with a Virtual Reality control panel on the lower right.

# Appendix A

## Mathematical Notations

In this chapter, specific (non-standard) mathematical notations that are used in this thesis are given.

### A.1 Sets of Standard Values

$\mathbb{B}$  is the set of boolean values  $\{true, false\}$ .

$\mathbb{N}$  is the set of natural numbers, excluding 0.

$\mathbb{N}_0$  is the set of natural numbers, including 0.

$\mathbb{Z}$  is the set of integers.

$\mathbb{R}$  is the set of real numbers.

$\mathbb{R}_+$  is the set of positive real numbers, excluding 0.

$\mathbb{R}_0^+$  is the set of non-negative real numbers, including 0.

### A.2 Power Sets

Power sets are denoted as  $\mathcal{P}(X)$  for sets  $X$ .

### A.3 Cardinality of Sets

The cardinality of a set  $X$  is denoted as:  $|X|$

### A.4 Element Selection

We use the notation

$$q = \mu(x : X \mid p)$$

for a set  $X$ , a bound variable  $x$  and a predicate  $p$  over  $x$  as a shorthand for  $|\{x \in X \mid p\}| = 1 \Rightarrow q \in \{x \in X \mid p\}$ . Further,

$$q = \mu(X)$$

is an abbreviation for  $q = \mu(x : X \mid true)$ .

## A.5 Functions

### A.5.1 Function Notations

The set of functions mapping elements of  $D$  to elements of  $R$  is denoted by

*Total Functions:*  $D \rightarrow R$

*Partial Functions:*  $D \dashrightarrow R$

*Finite Partial Functions:*  $D \dashrightarrow R$

*Injective Functions:*  $D \succrightarrow R$

*Injective Partial Functions:*  $D \succdashrightarrow R$

*Surjective Functions:*  $D \twoheadrightarrow R$

*Surjective Partial Functions:*  $D \dashrightarrow R$

*Bijjective Functions:*  $D \simeq R$

### A.5.2 Domain and Range of Functions

The domain of function  $f$  is denoted as:  $\text{dom } f$

The range of function  $f$  is denoted as:  $\text{ran } f$

### A.5.3 Inverse of a Function

The inverse of a bijective (total) function  $f$  is denoted as:  $f^{-1}$

### A.5.4 Overriding of Functions

A function  $f'$  can be defined by (partial) functions  $f$  and  $f_{ov}$ , such that  $f'$  coincides with  $f$ , with the difference specified by  $f_{ov}$ :

$$\begin{aligned} \forall (d \mapsto r) \in f_{ov} \bullet (d \mapsto r) \in f' \\ \forall (d \mapsto r) \in f \bullet d \notin \text{dom } f_{ov} \Rightarrow (d \mapsto r) \in f' \\ \text{dom } f' = \text{dom } f \cup \text{dom } f_{ov} \end{aligned}$$

Overriding is denoted by

$$f' = f \oplus f_{ov}$$

## A.6 Projection Functions

Projection functions for sets  $X = X_1 \times \dots \times X_n$  with  $1 \leq i \leq n$  are defined as

$$\begin{aligned} \pi_i X : X_1 \times \dots \times X_n \rightarrow X_i \\ (x_1, \dots, x_n) \mapsto x_i \end{aligned}$$

## A.7 Sequences

Finite sequences of elements of set  $X$  are given as partial finite functions from natural numbers to elements of  $X$ :

$$\text{seq } X = \{f : \mathbb{N} \rightharpoonup X \mid \text{dom } f = \{1..|f|\}\}$$

Sequences  $\{1 \mapsto x_1, \dots, n \mapsto x_n\}$  are abbreviated as  $\langle x_1, \dots, x_n \rangle$ . The empty sequence is denoted by  $\langle \rangle$ .

Concatenation of sequences  $\langle x_1, \dots, x_n \rangle$  and  $\langle y_1, \dots, y_n \rangle$  is written as

$$\langle x_1, \dots, x_n \rangle \hat{\ } \langle y_1, \dots, y_n \rangle = \langle x_1, \dots, x_n, y_1, \dots, y_n \rangle$$

The head of a non-empty sequence is given by

$$\text{head}(\langle x \rangle \hat{\ } s) = x$$

For any given set  $x \subseteq X$ , we assume an arbitrary, but fixed sequence of its elements, given by mappings

$$\text{anyseq}_X : \mathcal{P}(X) \rightarrow \text{seq } X$$

with

$$\forall x \subseteq X \bullet \text{ran}(\text{anyseq}_X(x)) = x \wedge |\text{anyseq}_X(x)| = |x|$$

Sequences of characters that act as conventional program strings are presented as

$$\text{"abc"} = \langle a, b, c \rangle$$

## A.8 Trees

We define two slightly different variants of trees for the use in this thesis. Both provide trees which contain a (root) node, along with a set of children trees, but for (1) *Unordered Trees*, the order of children *is not* significant, whereas for (2) *Ordered Trees*, the order of children *is*.

### A.8.1 Unordered Trees

Unordered trees are recursively defined as pairs of one node and one set of subtrees:

$$\text{tree } X = X \times \mathcal{P}(\text{tree } X)$$

The root node of an unordered tree is provided by

$$\text{node}_X = \pi_1 \text{ tree } X$$

The direct children nodes of the root node are accessed with

$$\text{children}_X : \text{tree } X \rightarrow \mathcal{P}(X)$$

$$(x, \{(x_0, t_0), \dots, (x_k, t_k)\}) \mapsto \{x_0, \dots, x_k\}$$

The set of all recursively contained subtrees, including the tree itself, is given by:

$$\text{subtrees}_X : \text{tree } X \rightarrow \mathcal{P}(\text{tree } X)$$

$$(x, \{t_0, \dots, t_k\}) \mapsto \{(x, \{t_0, \dots, t_k\})\} \cup \bigcup_{i=0}^k \text{subtrees}_X(t_i)$$

### A.8.2 Ordered Trees

Ordered trees are recursively defined as pairs of one node and one sequence of subtrees:

$$\text{tree}_o X = X \times \text{seq}(\text{tree}_o X)$$

The root node of an ordered tree is provided by

$$\text{node}_{o,X} = \pi_1 \text{tree}_o X$$

The direct children nodes of the root node are accessed with

$$\begin{aligned} \text{children}_{o,X} : \text{tree}_o X &\rightarrow \text{seq } X \\ (x, \langle (x_0, t_0), \dots, (x_k, t_k) \rangle) &\mapsto \langle x_0, \dots, x_k \rangle \end{aligned}$$

The set of all recursively contained subtrees, including the tree itself, is given by:

$$\begin{aligned} \text{subtrees}_{o,X} : \text{tree}_o X &\rightarrow \mathcal{P}(\text{tree}_o X) \\ (x, \langle t_0, \dots, t_k \rangle) &\mapsto \{(x, \langle t_0, \dots, t_k \rangle)\} \cup \bigcup_{i=0}^k \text{subtrees}_{o,X}(t_i) \end{aligned}$$

# Appendix B

## HybEL Grammar

In this chapter, the syntax of the HybridUML Expression Language is given by an EBNF grammar. It is provided for comprehensibility of the HybridUML Expression Language, and is somewhat imprecise. Particularly, the notion of the expression context from section 3.1 is omitted.

The applied EBNF “flavor” is taken from the ISO 14977 standard [Int96].

### B.1 Expressions

(\* Expressions are distinguished by their type. \*)

```
expression = boolExp | intExp | realExp | enumExp | sdtExp | diffExp
           | intSetExp | trigger | signalRaiseStmt | assignmentExpression ;
```

### B.2 Boolean Expressions

(\* Boolean expressions are one of

- (1) boolean literal
- (2) readable boolean variable
- (3) unary boolean operation
- (4) binary boolean operation (on boolean or numeric operands)
- (5) equality test operation (on boolean, numeric or enumeration-typed operands)
- (6) quantified expressions

\*)

```
boolExp = "(" , (
         "true" | "false"
         | readBoolVar
         | unaryBoolOp , boolExp
         | boolExp , binaryBoolOp , boolExp
         | numExp , num_x_num_to_bool_op , numExp
         | numExp , equality_op , numExp
         | enumExp , equality_op , enumExp
         | boolExp , equality_op , boolExp
```

```

| “∀” , intVar , “∈” , “{” , intSetExp , “}” , “•” , “(” , boolExp ,
“)”
| “∃” , intVar , “∈” , “{” , intSetExp , “}” , “•” , “(” , boolExp ,
“)”
) , “)” ;

```

### B.3 Numeric Expressions

(\* Integer expressions are one of  
(1) integer literal  
(2) readable integer variable  
(3) binary integer operation (on numeric operands)  
\*)

```

intExp = “(” , ( integerLiteral
| readIntVar
| intExp , num_x_num_to_num_op , intExp ) , “)” ;

```

(\* Real-valued expression are one of  
(1) real-valued literal  
(2) readable real-valued variable  
(3) binary real-valued operation (on numeric operands)

This includes constant expressions resulting from Analog Real variables as well.  
\*)

```

realExp = “(” , ( realLiteral
| readRealVar
| realExp , num_x_num_to_num_op , realExp
| intExp , num_x_num_to_num_op , realExp
| realExp , num_x_num_to_num_op , intExp ) , “)” ;

```

(\* Numeric expressions are integer expressions and real-valued expressions. \*)

```

numExp = intExp | realExp ;

```

### B.4 Enumeration-typed Expressions

(\* Enumeration-typed expressions are one of  
(1) enumeration-typed literal  
(2) readable enumeration-typed variable  
\*)

```

enumExp = enumLiteral | readEnumVar ;

```

### B.5 Structured Data Type Expressions

(\* Expressions of structured data type are one of  
(1) structured data type literal  
(2) readable variable of structured data type  
\*)

sdtExp = “{” , [ constExp , { “,” , constExp } ] , “}” | readSdtVar ;

(\* A shorthand for the above expressions, i.e. boolean, numeric, enumeration-typed and of structured data type, is defined. \*)

constExp = boolExp | numExp | enumExp | sdtExp ;

## B.6 Differential Expressions

(\* Differential expressions are real-valued expressions that contain at least one derivative of a variable. Differential expressions are one of

- (1) first derivative of a continuously readable variable
- (2) binary differential operation (on numeric or differential operands)

\*)

diffExp = “(” , ( readDerivVar  
 | diffExp , num\_x\_num\_to\_num\_op , diffExp  
 | diffExp , num\_x\_num\_to\_num\_op , realExp  
 | realExp , num\_x\_num\_to\_num\_op , diffExp  
 | diffExp , num\_x\_num\_to\_num\_op , intExp  
 | intExp , num\_x\_num\_to\_num\_op , diffExp ) , “)” ;

## B.7 Boolean Operations

(\* Unary boolean operation: negation. \*)

unaryBoolOp = “-” ;

(\* Binary boolean operations: (1) disjunction, (2) conjunction. \*)

binaryBoolOp = “∨” | “∧” ;

(\* Binary operations with boolean result: comparison for (1) equality, (2) inequality. \*)

equality\_op = “==” | “≠” ;

(\* Binary operations with boolean result on numeric operands: comparison for (1) less-than-or-equal-to relation, (2) less-than relation, (3) greater-than-or-equal-to relation, (4) greater-than relation. \*)

num\_x\_num\_to\_bool\_op = “<” | “≤” | “>” | “≥” ;

## B.8 Numeric Operations

(\* Binary numeric operations: (1) addition, (2) subtraction, (3) multiplication, (4) division, (5) raising to a power. \*)

num\_x\_num\_to\_num\_op = “+” | “-” | “.” | “/” | “^” ;

## B.9 Integer Set Expression

(\* Specification of finite integer sets: comma-separated integer value specifications, each is one of

- (1) one single constant integer expression
- (2) an (inclusive) range of integer values, specified by two constant integer expressions

\*)

intSetExp = intExp , “.” , intExp , [ “,” , intSetExp ] ;

## B.10 Assignment Expressions

(\* An assignment expression is either (1) a simple assignment, or (2) a quantified assignment. \*)

assignmentExpression = simpleAssignment | assignmentGroup ;

(\* A simple assignment assigns a single variable from a type-compatible expression. It is one of

- (1) boolean assignment
- (2) integer assignment
- (3) real-valued assignment
- (4) analog real assignment
- (5) enumeration-typed assignment
- (6) assignment to a variable of structured data type
- (7) assignment which includes the read or write access to the first derivative of an analog real variable

\*)

simpleAssignment = boolAss | intAss | realAss | analogrealAss | enumAss  
| sdtAss | diffAss ;

(\* A quantified assignment is a shorthand for a set of assignments, quantified by a bound integer variable. \*)

assignmentGroup = “∀” , intVar , “∈” , “{” , intSetExp , “}” , “:=” , “(”  
 , assignmentExpression , “)” ;

(\* A boolean assignment assigns a writable boolean variable. \*)

boolAss = boolVar , “:=” , boolExp ;

(\* An integer assignment assigns a writable integer variable. It is either of

- (1) “normal” assignment
- (2) non-deterministic assignment out of a set of integers, constrained by a constant boolean expression

\*)

intAss = intVar , “:=” , intExp  
| intVar , “∈” , “{” , intVar , “∈” , “{” , intSetExp , “}” , “|” ,  
boolExp , “}” ;

(\* A real-valued assignment assigns a real-valued variable that is discretely accessible. \*)

realAss = realVar , “:=” , numExp ;

(\* An analog real assignment assigns an analog real variable. \*)

analogrealAss = analogrealVar , “:=” , numExp ;

(\* An enumeration-typed assignment assigns a variable of enumeration type. \*)

enumAss = enumVar , “:=” , enumExp ;

(\* An assignment to a variable of structured data type assigns a value of structured data type. \*)

sdtAss = sdtVar , “:=” , sdtExp ;

(\* A differential assignment is one of

(1) assignment to the first derivative of an analog real variable

(2) assignment calculated from (at least one) derivative of an analog real variable

\*)

diffAss = writeDerivVar , “:=” , ( numExp | diffExp )  
 | analogrealVar , “:=” , diffExp ;

## B.11 Signal Raise Statements

(\* A signal raise statement is an expression that causes a signal to be sent. \*)

signalRaiseStmt = sendSig , [ “(” , [ constExp , { “,” , constExp } ] ,  
 “)” ] ;

## B.12 Trigger Expressions

(\* A trigger expression is an expression that defines a signal which can be received. \*)

trigger = recvSig , [ “(” , [ outParam , { “,” , outParam } ] , “)” ] ;

(\* Out-parameters define writable variables to store the data attached to signals, on reception of a signal, as defined by a trigger expression. The possible variables are

(1) boolean variable

(2) integer variable

(3) real-valued variable

(4) enumeration-typed variable

(5) variable of structured data type

\*)

outParam = boolVar | intVar | realVar  
 | analogrealVar | enumVar | sdtVar ;

## B.13 Variables

(\* For this grammar presentation, the check for type compatibility and for existence of a corresponding variable within the expression's contexts is omitted, therefore every variable is just a variable. \*)

```
constIntVar = var ;
constBoolVar = var ;
constRealVar = var ;
constAnalogrealVar = var ;
constEnumVar = var ;
constSdtVar = var ;
intVar = var ;
boolVar = var ;
realVar = var ;
analogrealVar = var ;
enumVar = var ;
sdtVar = var ;
```

(\* The variables that can be directly written by an expression are given by the corresponding identifiers. Additionally, analog real variables can be indirectly written, by use of their first derivative. This derivative is given by the variable's identifier, appended by an apostrophe. Alternatively, the identifier has a dot on top, which is the typical mathematical notation for derivatives wrt. time, but which is less convenient to denote by an EBNF grammar. \*)

```
writeDerivVar = analogrealVar , “ ’ ” ;
```

(\* The variables that can be read by an expression are given by the corresponding read-only variables, as well as the writable variables. \*)

```
readBoolVar = constBoolVar | boolVar ;
readIntVar = constIntVar | intVar ;
readRealVar = constRealVar | constAnalogrealVar
              | realVar | analogrealVar ;
readEnumVar = constEnumVar | enumVar ;
readSdtVar = constSdtVar | sdtVar ;
```

(\* The first derivative of constant or writable Analog Real variables can be read. This derivative is given by the variable's identifier, appended by an apostrophe. It can be represented graphically by a dot on top of the identifier. \*)

readDerivVar = constAnalogrealVar , “ ’ ” | writeDerivVar ;

(\* Variables, can be (1) simple identifiers, (2) simple identifiers with index expression, (3) sub-variables of a simple variable of structured data type, (4) or sub-variables of an indexed variable of structured data type. \*)

var = identifier | identifier , “[” , intExp , “]” | identifier , “.” , var | identifier , “[” , intExp , “]” , “.” , var ;

## B.14 Signals

(\* Sent signals are either (1) simple identifiers, or (2) simple indexed identifiers. \*)

sendSig = identifier | identifier , “[” , intExp , “]” ;

(\* Received signals are either (1) simple identifiers, or (2) simple indexed identifiers. A special index assignment expression can be applied to accept signals for all available indices, and then store the actual received index. \*)

recvSig = identifier | identifier , “[” , ( intExp | “:=” , intVar ) , “]” ;

## B.15 Literals

(\* For numeric literals, non-empty sequences of digits are provided. \*)

digits = ( “0” | “1” | “2” | “3” | “4” | “5” | “6” | “7” | “8” | “9”  
), [ digits ] ;

(\* Integer literals are a non-empty sequence of digits, optionally with sign. \*)

integerLiteral = [ “-” ] , digits ;

(\* Real-valued literals consist of two non-empty sequences of digits, separated by a dot, and an optional sign. \*)

realLiteral = [ “-” ] , digits , “.” , digits ;

(\* Literals for enumeration-typed values consist of the type and the literal value, separated by a double colon. \*)

enumLiteral = identifier , “:.” , identifier ;



## Appendix C

# Case Study: Radio-Based Train Control

This chapter provides the full HybridUML model of the case study “Radio-Based Train Control”. For better comprehensibility, a summary of the original case study description is contained.

### C.1 Case Study Description: Radio-Based Train Control

The case study “Radio-Based Train Control” provided in the context of the DFG priority programme Software Specification [DFG], is derived from a respective requirements specification [FFB96] of the Deutsche Bahn AG (German Railways).

In this section, we *cite* the summarized case study description from [HPSS04], in order to guarantee the comprehensibility of the HybridUML model presented in section C.2:

**Regular Behavior Definition.** The problem chosen for the reference case study is a decentralized radio-based control system for a railway level crossing, where a single track railway line and a road cross on one level. The intersection area of the road and the railway line is called the danger zone, since trains and road traffic are not allowed to enter it at the same time to avoid collision. The railway crossing is equipped with barriers and road traffic lights. Traffic lights at the level crossing consist of a red and a yellow light. When the yellow light is shown road users (drivers, cyclists, pedestrians etc.) should stop at the level crossing if possible. The red light means that the level crossing is closed for road traffic and road users are not allowed to enter it. The yellow and red lights must never be on together. When both lights are off, road users may enter the crossing area. Half arm barriers are used to block the entry lane on either side of the level crossing. Since there are no barriers for the exit lanes, road users may possibly enter the crossing area on the opposite lane. Although this behaviour constitutes a severe contravention of the traffic regulations, it can be frequently observed due to long waiting times at closed level crossings. This has to be taken into account for the level crossing control system by monitoring a maximum closure time.

The traffic lights and barriers at the level crossing are controlled by the level crossing control system. It is activated when a train is approaching the level crossing. In the activated mode the level crossing control system performs a sequence of actions according to a specific timing in order to safely close the crossing and to ensure the danger zone is free of road traffic. First, the traffic lights are switched on to show the yellow light; then after 3 seconds they are switched to red. Approximately 9 seconds later, the barriers start to be lowered. If the barriers have been completely lowered within a maximum time of 6 seconds, the level crossing control system signals that the level crossing is in its safe state, thus allowing the train to pass the level crossing. When the train has completely passed the crossing area the level crossing may be opened for road traffic again and the level crossing control system switches back to the deactivated mode.

The main components of interest for software specification are the train-borne control system, the trackside level crossing control system and the operations centre. These main components can communicate with each other by means of mobile radio communication. Transmission times on the radio network may vary and have to be considered. Radio telegrams even may get lost on the radio network.

The approaching of a train at the level crossing is traditionally detected by trackside equipment or signal staff such that the level crossing can be closed in time to let the train pass through without any delay or braking action. In modern radio-based train control systems the activation of the level crossing is based on continuous self-localization of the train and mobile communication between the train and the decentralized level crossing control system. A route map on board the train contains the positions of potential danger points at level crossings and provides additional information for the train on when or where to send an activation order to the respective level crossing control system.

Self-localization is realized by balises, i.e. small transponders between the rails, transmitting an identification signal to the train. Comparison of this information with the digital route map stored on board the train allows an exact positioning of the train.

When the on-board system detects that the train is approaching a level crossing it sends a radio message to the level crossing control system to switch on the road traffic lights and to lower the crossing barriers. It will also set a braking curve for the speed supervision system, which will make the train stop at the potential danger point in a failure situation. The level crossing control system acknowledges receipt of the activation order to the train. After receipt of the acknowledgment the on-board system waits an appropriate time for the level crossing to be closed and then sends a status request to the level crossing control system. If the level crossing is in its safe state, this will be reported to the train. This allows the train to cancel the braking curve and safely pass over the level crossing while supervising the regular speed profile. The triggering of the vehicle sensor at the end of the level crossing will cause the barriers to be opened again and the traffic lights to be switched off.

**Failure Behavior Definition.** Possible failure conditions have to be taken into account to achieve safe control of the level crossing and the train. A main cause of failures is the malfunctioning of sensors or actuators. Faults may also occur in the main physical structures. Failures of communication systems may affect the communication between control

systems and devices as described above for radio networks and mobile communication. Last, but not least the control systems themselves may fail.

Defective devices will be repaired after some time so that the occurrences of both failures and repairs have to be taken into account. While failures may occur at any time, repair of defective devices in the case of non-recoverable failures, mostly physical components or sensors and actuators, will not take place when there is a train approaching or passing the level crossing.

In the case study only a limited number of failures are considered: failures of the yellow or red traffic lights (to be considered separately), the barriers, the vehicle sensor and the delayed receipt or loss of telegrams on the radio network. The traffic lights and the vehicle sensor are constantly monitored and defects are immediately reported to the level crossing control system. Failure of the barriers can only be detected by time-out when barriers fail to reach the upper or lower end position in time or at all. The required behaviour of the control systems under failure conditions will be described below according to the time sequence of failure occurrences and control reactions.

The level crossing control system is able to detect the occurrence of failures of traffic lights and vehicle sensors. It immediately reports such an event to the operations centre, which is able to have the defective component repaired. After repair, it may be necessary to carry out re-initialisation of the level crossing control system (e.g. barrier failure). This does not imply that train operation is suspended on the affected section of track for the time up until the repair is carried out.

After having sent the activation order to the level crossing, the train waits for an acknowledgment. The train will send no status request until the acknowledgment has been received. The following applies in all cases, whether the train has sent the status request having received the acknowledgment or not. If the train does not subsequently receive the status report indicating that the level crossing is in its safe state before entering its breaking curve, the on-board system will apply the brakes until the status report has been received or the train has come to a standstill. If the status report is received before the train comes to a complete stop, the brakes are released and the train can continue. Otherwise the system causes a message to appear on the driver's display, asking the driver to make sure that it is safe to cross the level crossing and to give confirmation via the display unit that the level crossing is in its safe state. If meanwhile the status report has been received, the message is cancelled from the display, the brakes are released and the driver does not need to give confirmation. Otherwise the driver has to confirm that the level crossing is in its safe state in order to release the brakes and continue the journey.

After receipt of an activation order from a train, the level crossing control system immediately checks if the level crossing control should be activated, and accordingly will or will not send an acknowledgment to the train. The level crossing control system will not be activated if the red traffic lights or the vehicle sensor are defective. If the level crossing control system has been activated, then, after a minimum green time has passed since the last deactivation of the level crossing, the yellow traffic light is switched on for 3 seconds. If the yellow traffic light becomes defective either before

or during the yellow light period, the traffic lights are switched to red and the red light period of 9 seconds is extended accordingly by the lost yellow light time. If the red traffic light fails after activation of the level crossing control system the closing procedure has to be cancelled unless the lowering of the barriers has already begun. The level crossing must be reported to be in the failure state if the barriers fail to be completely lowered within 6 seconds from the start of lowering or if in the meantime the red traffic light has become defective. Upon request, the current status of the level crossing will be reported to the train.

If the vehicle sensor becomes defective, the level crossing control system cannot be deactivated anymore by a passing train. Accordingly the barriers remain lowered and the red traffic light remains switched on. However, the level crossing control system monitors a maximum closure time of 240 seconds starting from the time the red lights are switched on. After the maximum closure time has elapsed, no positive status report is sent to the train. The level crossing control system will report the exceeding of the maximum closure time to the operations centre. The operations centre finds out, whether the train has already passed the level crossing or not. In the first case, the operations centre sends a deactivation order to the level crossing. Otherwise the train is still approaching or just passing over the level crossing and the rules for late arrival at the level crossing apply as described above.

Regarding redundancies and symmetries within the geometry and equipment at the level crossing, it was recommended that any multiplicity of devices or processes should be ignored (e.g. number of trains, level crossings or directions of train traffic). For the purpose of specifying non-finite behaviour, the track may be assumed to be virtually circular. However a clear separation of different laps made on the track is recommended, so that no two successive closing procedures of the level crossing overlap. Therefore it might be necessary to synchronize e.g. the opening of the barriers with the next sending of an activation order by the train. Also note that the train driver may slow down, stop or speed up the train at any moment or location. This must not affect the safety of the level crossing control system. The train may be assumed to always run in the same direction. The described reference case study combines different aspects of specification problems from the traffic control system domain and was successfully used for a comparison of properties and practical capabilities of different formal specification techniques.

□

## C.2 HybridUML model: Radio-Based Train Control

### C.2.1 System

: System ((C.1) actprmGC, (C.2) actprmRA)

Figure C.1: Top-level System Instance.

(C.1) **actprmGC**  $\equiv$   
 {3, 11, 200, -2, 5.6, 3, 2, 2, 8, 1,  
 -0.1, 0.1, 90, 0, -10, 10, -15, 15, 15, 6,  
 6, 180, 8000, 60, 30, 1.5, 20, 100, 7, 100,  
 27, 100, 17, 7}

(C.2) **actprmRA**  $\equiv$   
 {  
 {0, 27.8, *VTP\_TYPE::VELOCITY\_CHANGE*, 0, 0},  
 {1600, 16.7, *VTP\_TYPE::VELOCITY\_CHANGE*, 1, 0},  
 {3000, 33.3, *VTP\_TYPE::VELOCITY\_CHANGE*, 2, 0},  
 {4200, 30.5, *VTP\_TYPE::VELOCITY\_CHANGE*, 3, 0},  
 {5200, 11.1, *VTP\_TYPE::VELOCITY\_CHANGE*, 4, 0},  
 {5900, 9.7, *VTP\_TYPE::VELOCITY\_CHANGE*, 5, 0},  
 {6400, 22.2, *VTP\_TYPE::VELOCITY\_CHANGE*, 6, 0},  
 {7100, 16.7, *VTP\_TYPE::VELOCITY\_CHANGE*, 7, 0},  
 {2800, 0, *VTP\_TYPE::CROSSING*, 8, 0},  
 {4960, 0, *VTP\_TYPE::CROSSING*, 9, 1},  
 {7205, 0, *VTP\_TYPE::CROSSING*, 10, 2},  
 {2800, 2840, 6, 12, 13, 0, 8},  
 {4960, 5000, 6, 12, 13, 1, 9},  
 {7205, 7245, 6, 12, 13, 2, 10}  
 }

This defines a route with 11 velocity target points and 3 crossings. Each velocity target point has an absolute position on the track, an associated maximum velocity, a type, an identifier, and optionally an associated crossing identifier. The crossings are defined by a beginning and an end position, some timing parameters, an identifier, and the identifier of the corresponding velocity target point.

```

<<agent>>
System (GC:GlobalConstants, RA:RouteAtlas)

```

Figure C.2: Agent System.

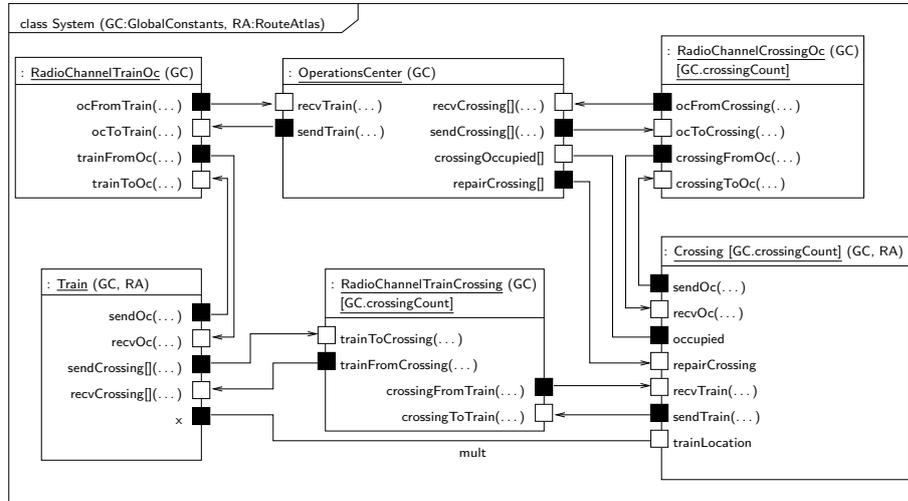


Figure C.3: Structure of System.

The complete system consists of:

*Train* One train.

*Crossings* A fixed number of railroad crossings.

*Operations center* One operations center.

*Radio channels* Radio channels connecting the train and the crossings, the crossings and the operations center, and the operations center with the train.

The controller components within the train, the crossings and the operations center communicate via radio channels, therefore each entity has respective *send...* and *rcv...* signals. They are mapped to *...From...* and *...To...* signals of the corresponding radio channel. The radio channel internally models the transmission of the radio telegram, such that an incoming *...From...* signal causes an appropriate *...To...* signal (if the telegram is not lost).

Since there are multiple crossings and associated radio channels, each shared signal or variable is mapped *point-to-point* – either by agent index or by signal index. In contrast, the train’s location is *multicasted*, i.e. one single variable is shared for the train and all connected crossings.

Implicitly, there is one simple track without branches. Thus every position is specified by a distance relative to the track’s beginning. Every crossing has a fixed beginning and ending on the track; the train’s position is always on the track, too. As soon as the train passes the end of the track, it is located at its beginning again; therefore the track can be thought of being circular.

### C.2.2 Data Types

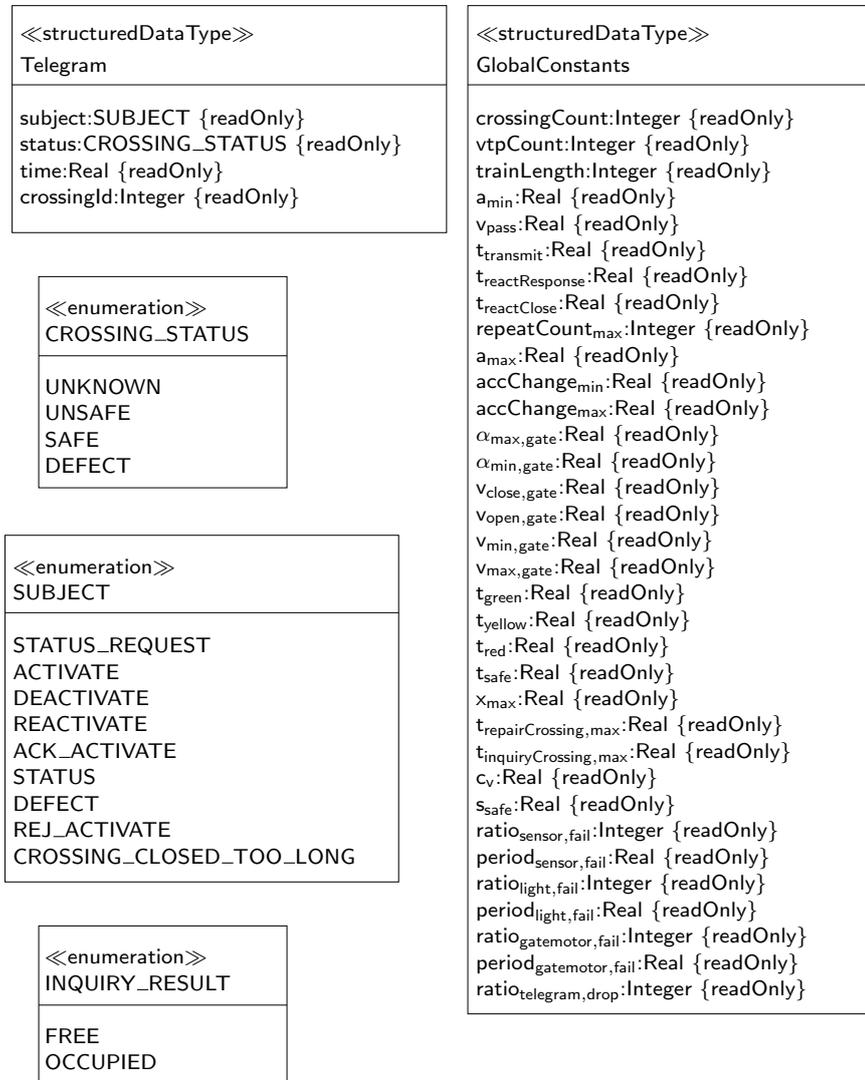


Figure C.4: Data Types.

These data types are used by the train controller as well as the crossing controllers:

**Telegram** This is the top-level data structure for radio telegrams. It consists of a mandatory subject and optionally of a crossing status, a time value and a crossing identifier:

**subject** Denotes the meaning of the telegram.

**status** If subject = SUBJECT::STATUS, then this must contain the status of the sending crossing.

**time** If  $\text{subject} = \text{SUBJECT}::\text{STATUS} \wedge \text{status} = \text{CROSSING\_STATUS}::\text{SAFE}$ , then this must be the remaining safe close duration of the sending crossing. Unit is: *s*

**crossingId** If  $\text{subject} = \text{SUBJECT}::\text{DEFECT}$ , then this must be the identifier of the defect crossing.

**CROSSING\_STATUS** The possible crossing statuses are:

**UNKNOWN** The status is unknown.

**UNSAFE** The crossing is not safe.

**SAFE** The crossing is safe. This is the only status which allows the train to pass.

**DEFECT** The crossing is defect.

**SUBJECT** Radio telegrams have on of these subjects:

**STATUS\_REQUEST** This subject denotes a request for status information, sent from the train to a crossing.

**STATUS** This subject denotes contained status information about a crossing. Any crossing sends this to the train.

**ACTIVATE** This subject is contained in a request for activation (i.e. closing) from the train to a crossing.

**REACTIVATE** This subject is synonymic to **ACTIVATE**.

**ACK\_ACTIVATE** This denotes the positive receipt of a crossing to the train meaning that the crossing starts closing.

**REJ\_ACTIVATE** This denotes the negative receipt of a crossing to the train meaning that the crossing will not start closing.

**DEACTIVATE** A telegram with this subject is sent to a crossing in order to deactivate (i.e. open) it.

**DEFECT** This subject is used for telegrams to the operations center in order to inform it about a crossing defect.

**CROSSING\_CLOSED\_TOO\_LONG** This subject is used for telegrams to the operations center in order to inform it about a violation of the crossing's maximum safe close time duration.

**GlobalConstants** This data structure is a collection of constants used in several parts of the specification:

**crossingCount** The number of crossings within the complete system.

**vtpCount** The number of velocity target points within the complete system.

**trainLength** This is the length of the train. Unit is: *m*

- $a_{\min}$  This is the minimum acceleration (i.e. the maximum deceleration) of the train. Unit is:  $\frac{m}{s^2}$
- $a_{\max}$  This is the maximum acceleration of the train. Unit is:  $\frac{m}{s^2}$ .
- $v_{\text{pass}}$  See agent BrakePointController in section C.2.17 for an explanation. Unit is:  $\frac{m}{s}$
- $t_{\text{transmit}}$  This is the maximum time that is needed for the transmission of a radio telegram.
- $t_{\text{reactResponse}}$  This is the (assumed) maximum reaction time of a crossing before responding to an activation request or a status request. Unit is:  $s$
- $t_{\text{reactClose}}$  This is the (assumed) maximum reaction time of a crossing before closing after the reception of an activation request. Unit is:  $s$
- $\text{repeatCount}_{\max}$  This defines the maximum number of tries that a train sends an activation request or status request before assuming that the crossing is defect, if the crossing does not respond.
- $\text{accChange}_{\min}$  The minimum change rate (i.e. the maximum negative change rate of the (user-requested) acceleration. See also section C.2.4. Unit is:  $\frac{m}{s^3}$
- $\text{accChange}_{\max}$  The maximum change rate (i.e. the maximum negative change rate of the (user-requested) acceleration. See also section C.2.4. Unit is:  $\frac{m}{s^3}$
- $\alpha_{\max, \text{gate}}$  Maximum angle of the gate barriers; i.e. the barrier angle when open. Unit is:  $1^\circ$
- $\alpha_{\min, \text{gate}}$  Maximum angle of the gate barriers; i.e. the barrier angle when closed. Unit is:  $1^\circ$
- $v_{\text{close, gate}}$  Angular velocity that is applied by the gate motor to the gate barrier when closing the barrier, if the motor is operative. Unit is:  $\frac{1^\circ}{s}$
- $v_{\text{open, gate}}$  Angular velocity that is applied by the gate motor to the gate barrier when opening the barrier, if the motor is operative. Unit is:  $\frac{1^\circ}{s}$
- $v_{\min, \text{gate}}$  Minimum angular velocity that the motor can apply to the gate barrier. That is the *fastest way to close* it. Unit is:  $\frac{1^\circ}{s}$
- $v_{\max, \text{gate}}$  Maximum angular velocity that the motor can apply to the gate barrier. That is the *fastest way to open* it. Unit is:  $\frac{1^\circ}{s}$
- $t_{\text{green}}$  Minimum time duration that a crossing must provide to road users after opening, in order to cross it. Unit is:  $s$
- $t_{\text{yellow}}$  Time duration that a crossing's yellow light is active while activating the crossing. Unit is:  $s$
- $t_{\text{red}}$  Time duration that a crossing's red light is active while activating the crossing. Unit is:  $s$

- $t_{\text{safe}}$  Time duration that a crossing is assumed to be safe after successful activation. When this duration is exceeded, the crossing is assumed to be unsafe: road users are assumed to bypass the gates when becoming impatient. Unit is:  $s$
- $x_{\text{max}}$  The end position of the (circular) track. This is where it is connected with its beginning position 0. Unit is:  $m$
- $t_{\text{repairCrossing,max}}$  Maximum time duration that is needed to repair a crossing, measured since the request is received by the operations center. Unit is:  $s$
- $t_{\text{inquiryCrossing,max}}$  Maximum time duration that is needed to inquire a crossing, measured since the request is received by the operations center. Unit is:  $s$
- $c_v$  *Velocity Coefficient* – this is a relative tolerance used for velocity-based calculations like braking distance. It is used as a factor for the velocity. Values  $c_v \geq 1$  shall be applied, increasing the assumed braking distance.
- $s_{\text{safe}}$  *Safety Distance* – this is an absolute tolerance used for distance calculations like braking distance. It is simply added. Values  $s_{\text{safe}} > 0$  shall be used. Unit is:  $m$
- $\text{ratio}_{\text{sensor,fail}}$  This defines a probability  $p$  for which the sensor fails when it is possible:  $p = \frac{1}{\text{ratio}_{\text{sensor,fail}}}$ . See also  $\text{period}_{\text{sensor,fail}}$ .
- $\text{period}_{\text{sensor,fail}}$  This models a frequency for *potential* failures of the switch-off sensors. Once for each period, it is determined with a certain probability, whether the sensor fails. The probability is given by  $\text{ratio}_{\text{sensor,fail}}$ . Unit is:  $s$   
For example,  $\text{period}_{\text{sensor,fail}} = 5 \wedge \text{ratio}_{\text{sensor,fail}} = 6$  models one sensor failure per 30 seconds on average.
- $\text{ratio}_{\text{light,fail}}$  Analogical to  $\text{ratio}_{\text{sensor,fail}}$ , wrt. failures of lights.
- $\text{period}_{\text{light,fail}}$  Analogical to  $\text{period}_{\text{sensor,fail}}$ , wrt. failures of lights.
- $\text{ratio}_{\text{gatemotor,fail}}$  Analogical to  $\text{ratio}_{\text{sensor,fail}}$ , wrt. failures of gate motors.
- $\text{period}_{\text{gatemotor,fail}}$  Analogical to  $\text{period}_{\text{sensor,fail}}$ , wrt. failures of gate motors.
- $\text{ratio}_{\text{telegram,drop}}$  Analogical to  $\text{ratio}_{\text{sensor,fail}}$ , wrt. failures of telegram transmissions between train and crossing. In contrast to sensors, no period is modeled; the decision if a telegram is dropped or successfully delivered is made exactly once.

**INQUIRY\_RESULT** In case of a (manual) crossing inquiry, there are possible inquiry results:

FREE The crossing is free.

OCCUPIED The crossing is occupied by the train.

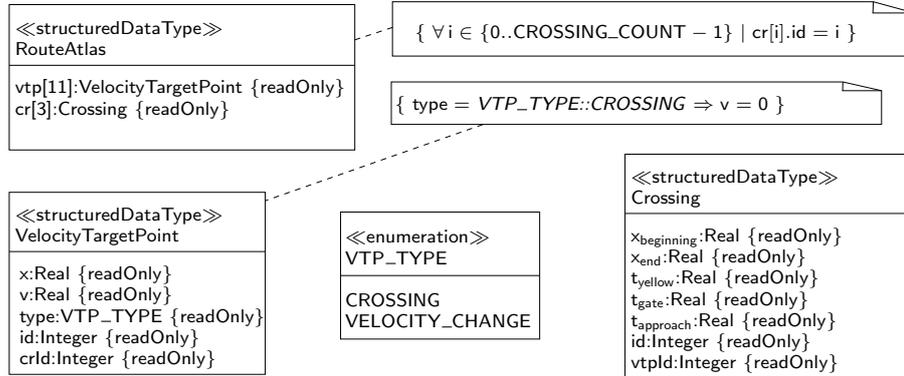


Figure C.5: Data Types.

These are train controller specific data types:

**RouteAtlas** The route atlas has associated velocity target point instances and associated crossing instances:

- vtp** For each velocity target point on the track, there is an instance of **VelocityTargetPoint**. The multiplicity shall correspond to the number of velocity target points, which is defined by `GlobalConstants::vtpCount`. Unfortunately, there are no parameters for structured data types, therefore the multiplicity must be explicitly given here.
- cr** For each crossing on the track, there is an instance of **Crossing**. The multiplicity shall correspond to the number of crossings, which is defined by `GlobalConstants::crossingCount`. It must be explicitly given here, like the multiplicity of **vtp**.

Since each crossing is associated with one velocity target point, the following holds:  $\exists VC\_COUNT \bullet VC\_COUNT = VTP\_COUNT - CROSSING\_COUNT \geq 0$  and thus  $VTP\_COUNT \geq CROSSING\_COUNT$ .

**VelocityTargetPoint** A velocity target point denotes a location on the track at which the train must not exceed a defined maximum velocity. It either is associated with a crossing, then there is a distinction between active and inactive,<sup>1</sup> or it represents a restrictive velocity change from the static velocity profile. The latter are always active.

- x** This is the location of the velocity target point on the track. Unit is:  $m$
- v** This is the maximum allowed velocity. If the train reaches this velocity target point, its velocity must not be greater.  
If this velocity target point is associated with a crossing, then  $v = 0$ , i.e. the train must stop (if the point is active). Unit is:  $\frac{m}{s}$
- type** The type of velocity target point is either `VTP_TYPE::CROSSING` or `VTP_TYPE::VELOCITY_CHANGE` as described above.

<sup>1</sup>See section C.2.11: `vtpActive`.

- id Each velocity target point has a unique identifier.
  - crld This is the identifier of the associated crossing, iff `type = VTP_TYPE::CROSSING`. Otherwise its value is not meaningful.
- VTP\_TYPE** The velocity target point types are:
- CROSSING The velocity target point is associated with a crossing.
  - VELOCITY\_CHANGE The velocity target point denotes a restrictive velocity change.
- Crossing** This data structure represents a crossing on the track.
- $x_{\text{beginning}}$  This is the location of the crossing's beginning on the track. Unit is:  $m$
  - $x_{\text{end}}$  This is the location of the crossing's end on the track. Unit is:  $m$
  - $t_{\text{yellow}}$  Time duration that a crossing's yellow light is active while activating the crossing. Unit is:  $s$
  - $t_{\text{gate}}$  This is the time duration that is needed to close the gates of this crossing. Unit is:  $s$
  - $t_{\text{approach}}$  This is the maximum time duration an approaching road user needs to either stop before the crossing or completely pass it. Unit is:  $s$
- id Each crossing has a unique identifier.
  - vtpld This is the identifier of the associated velocity target point.

### C.2.3 Train

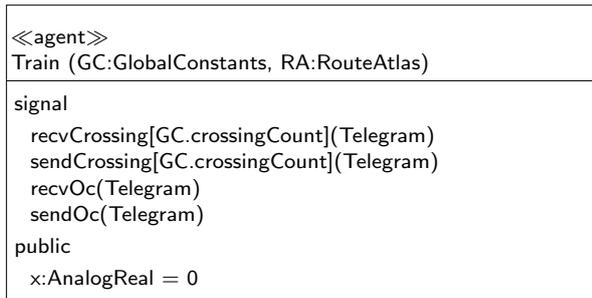


Figure C.6: Agent Train.

The train always has a position  $x$  relative to the beginning of the track – this is the location of its head.

Communication with the operations center and with the crossings is accomplished by the sending and receiving of radio telegrams.

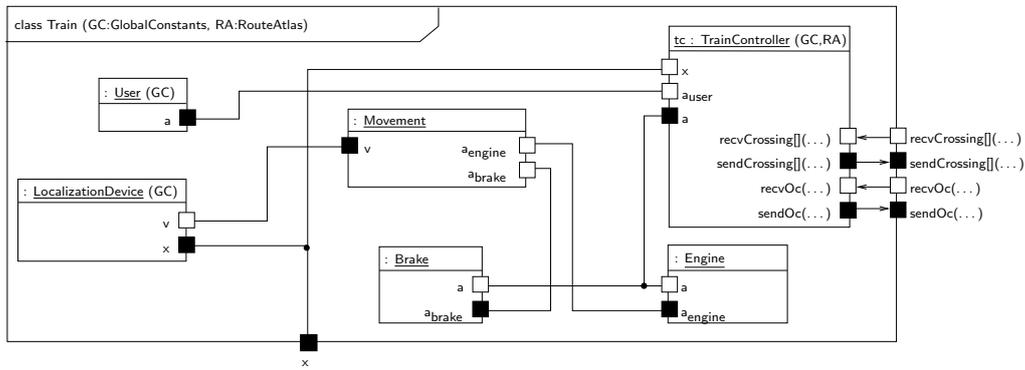


Figure C.7: Structure of Train.

The train consists of:

*Engine* The engine is responsible for the (positive) acceleration of the train.

*Brake* The brake is responsible for the deceleration of the train.

*Localization device* The localization device determines the train’s actual position.

*Movement* The train’s movement defines the speed of the train on the basis of its acceleration.

*Controller* The train controller is the train’s controlling hardware and software.

*User* The locomotive driver.

The train controller *tc* provides a target acceleration *a* that is realized by the engine and the brake. Both provide an acceleration each, their sum is the actual acceleration of the train. From this, the train’s current velocity *v* results (agent *Movement*), which in turn implies the location of the train (sensed by the *LocalizationDevice*). The current location is feedback to the train controller again, additionally the locomotive driver (*User*) provides a user-requested acceleration. In order to communicate with the crossing controllers and the operations center controller, signal interfaces for sending and receiving radio telegrams are provided. The train’s length and position are accessible from the environment because crossings (particularly the train sensors) must be able to detect the train.

### C.2.4 User

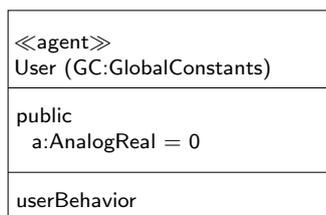


Figure C.8: Agent User.

The user provides a user-requested acceleration  $a$ .  
This is defined in top-level mode `userBehavior`.

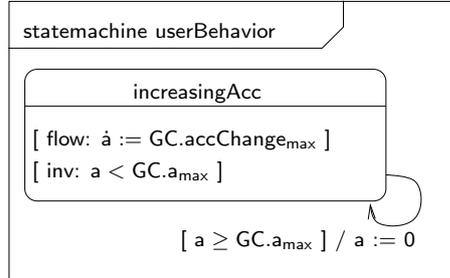


Figure C.9: Behavior of User.

The user chooses the required acceleration within these bounds:

$GC.a_{max}$  This is the maximum acceleration that may be requested by the user.

$GC.accChange_{max}$  This is the maximum change of the user-requested acceleration.

The user is assumed to constantly increase the requested acceleration up to  $GC.a_{max}$ . When  $GC.a_{max}$  is reached, the user restarts with no acceleration.

### C.2.5 LocalizationDevice



Figure C.10: Agent LocalizationDevice.

The localization device senses the train's location  $x$  on the track. It is assumed that the location is consistent with the progression (since initialization of the complete system) of the velocity  $v$  of the train. The value  $x_{max}$  defines the end of the track, which is also the beginning because a circular track is expected: when the train reaches  $x_{max}$ , it is (by definition) at location 0 again.

`localizationDeviceBehavior` defines this behavior.

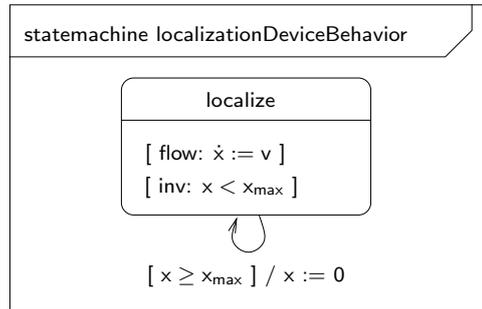


Figure C.11: Behavior of LocalizationDevice.

It is assumed that the location is consistent with the progression (since initialization of the complete system) of the velocity  $v$  of the train, thus the first derivative of  $x$  with respect to time is just the velocity  $v$ . If the end of the track  $x_{\max}$  is reached, it is (by definition) at location 0 again.

### C.2.6 Movement

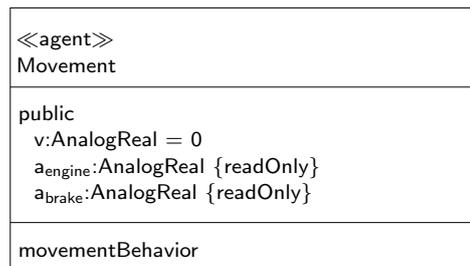


Figure C.12: Agent Movement.

The movement agent defines the current velocity  $v$  of the train that depends on the accelerations  $a_{\text{engine}}$  and  $a_{\text{brake}}$  provided by the engine and the brakes.

Its behavior is defined by `movementBehavior`.



Figure C.13: Behavior of Movement.

The change of the current train velocity  $v$  is its acceleration, that in turn is the sum of the accelerations  $a_{\text{engine}}$  and  $a_{\text{brake}}$  provided by the engine and the brakes.

### C.2.7 Brake

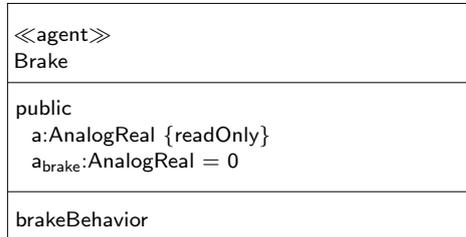


Figure C.14: Agent Brake.

The train brake reads the requested acceleration  $a$  from the train controller and applies a deceleration  $a_{brake}$  as defined in top-level mode `brakeBehavior`.

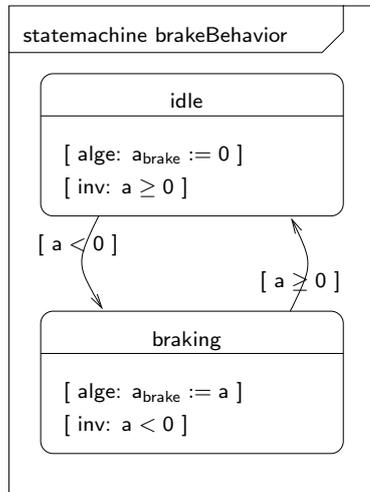


Figure C.15: Behavior of Brake.

As long as the controller requested acceleration  $a$  is negative, the train brakes with exactly this deceleration. Otherwise, it does not brake.

### C.2.8 Engine

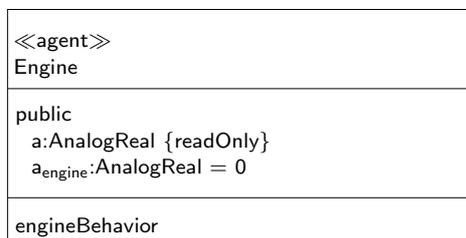


Figure C.16: Agent Engine.

The train’s engine realizes the positive acceleration  $a_{engine}$  according to the target acceleration  $a$ .

This is defined in `engineBehavior`.

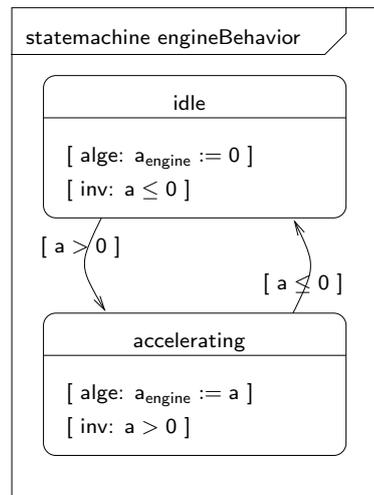


Figure C.17: Behavior of Engine.

This state machine sets the engine's part (i.e. the positive part) of the train's acceleration:  $a_{\text{engine}}$ . If the target acceleration  $a$  is positive, then the engine realizes it completely and therefore the brakes must not be active (mode **accelerating**), otherwise the engine does not provide any acceleration and the brakes are assumed to realize the deceleration (mode **idle**).

### C.2.9 TrainController

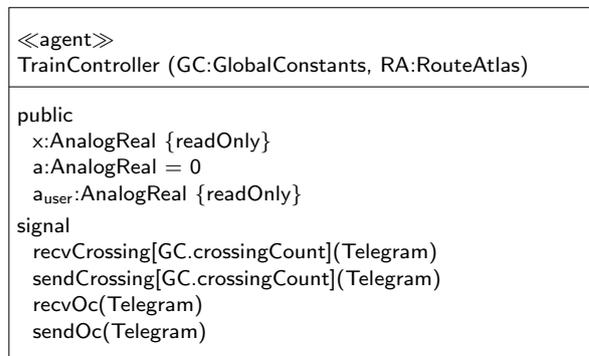


Figure C.18: Agent TrainController.

The train controller's purpose is to continuously determine the target acceleration  $a$  that subsumes positive acceleration as well as braking. This is controlled by the user-requested acceleration  $a_{\text{user}}$  in combination with several constraints that follow from velocity restrictions contained in the route atlas  $RA$  and from crossing statuses that are received via radio telegrams (**recv...**). Further, the train triggers the closing of crossings via sending radio telegrams (**send...**). In order to accomplish this, the actual location  $x$  of the train must be provided by the environment.

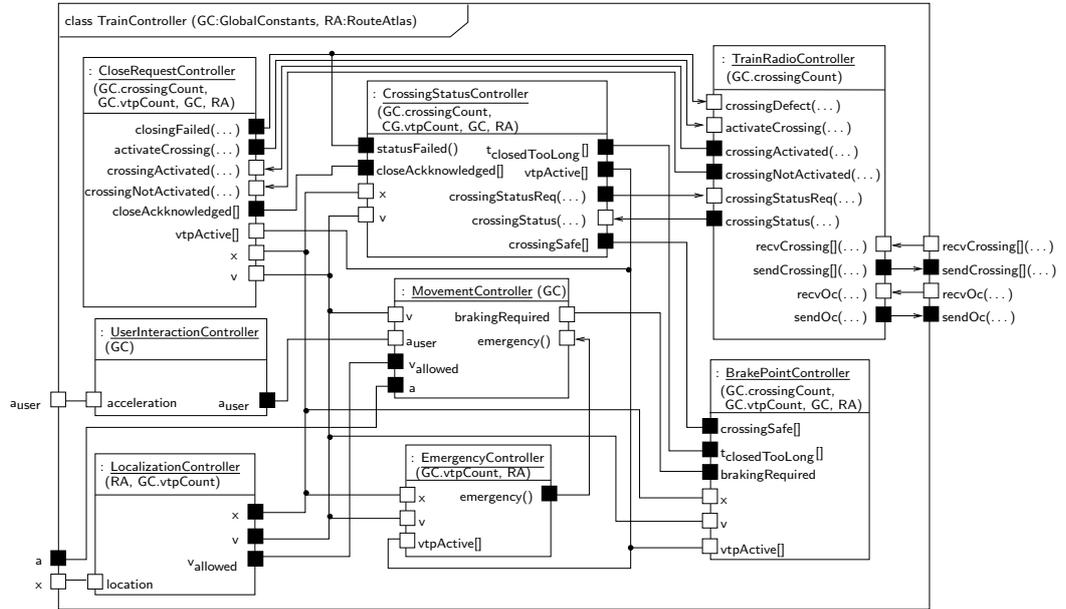


Figure C.19: Structure of TrainController.

The train controller consists of several parallel entities:

*Brake point controller* The brake point controller continuously checks if enforced braking is necessary. This is the result of approaching an active velocity target point too fast.

*Emergency controller* The emergency controller detects catastrophic situations like passing a velocity target point too fast, emitting an emergency signal.

*Movement controller* The movement controller actually sets the acceleration, particularly taking the brake point controller’s enforced braking recommendation and the emergency signal into account.

*User interaction controller* This controller adjusts the user requested acceleration within a possible acceleration range.

*Close request controller* The close request controller’s responsibility is to request the closing of crossings that the train approaches.

*Crossing status controller* The crossing status controller requests and keeps track of the crossings’ statuses that the train approaches.

*Localization controller* This controller keeps track of the train’s position on the track. It also provides its current velocity.

*Train radio controller* This agent is the radio communication entity.

### C.2.10 CloseRequestController

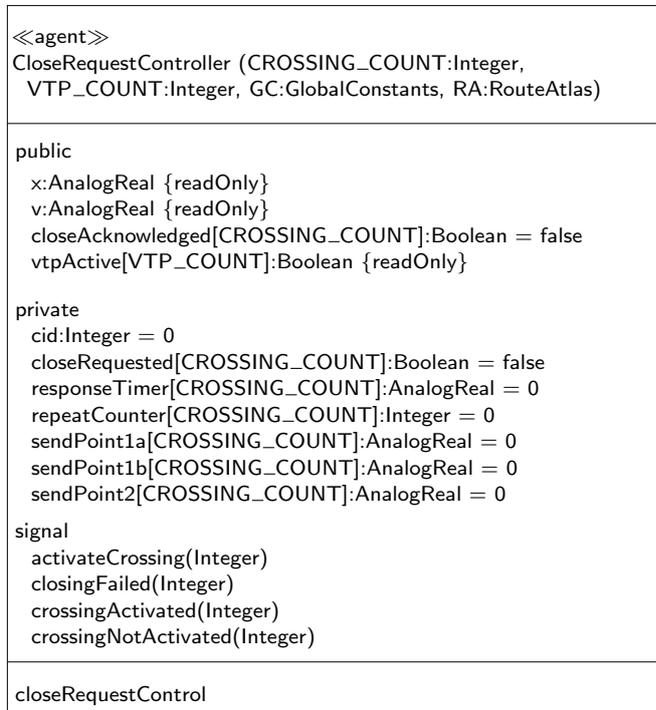


Figure C.20: Agent CloseRequestController.

This agent defines the management of close requests that are sent to crossings that the train approaches:

In order to decide if a close request has to be sent, the controller needs information about the crossings (from the route atlas RA), as well as the current location  $x$  and velocity  $v$  of the train. For each velocity target point a flag denotes if it is active or not (and therefore if the associated crossing is unsafe or safe).

If a close request is needed, the signal `activateCrossing(...)` is sent. Possible results are received via `crossingActivated(...)` and `crossingNotActivated(...)`. `closingFailed(...)` is emitted, if the crossing is not successfully closed.

Several internal variables are used to store the request status, to check time-outs and to count the number of tries. Two kinds of send points are continuously calculated for the determination of the request necessity.

The variables `closeAcknowledged` contain the confirmations of successful close request results. They are reset if the corresponding crossing has been passed by the train.

The behavior of the close request controller is defined in top-level mode `closeRequestControl`.

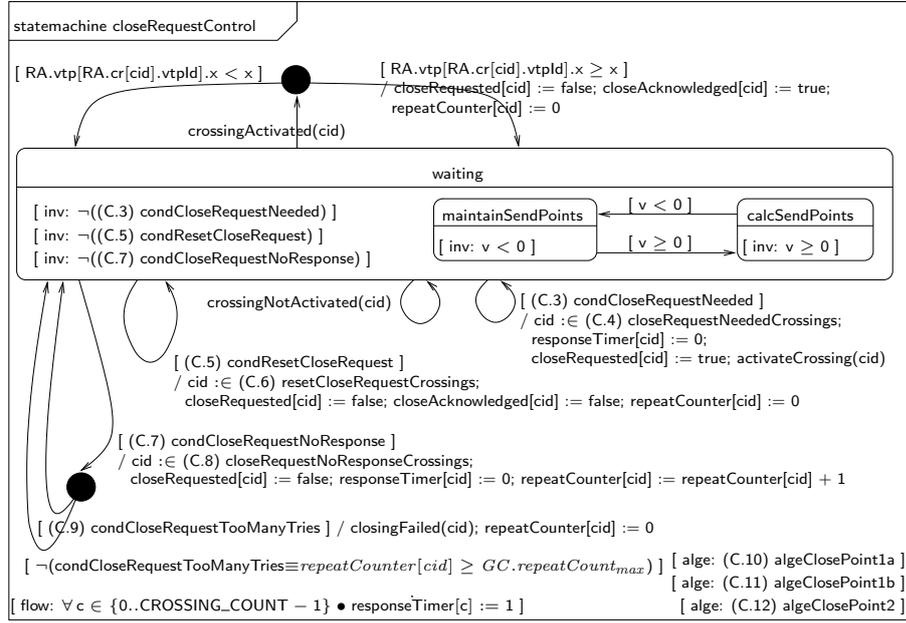


Figure C.21: Behavior of CloseRequestController.

- (C.3) **condCloseRequestNeeded**  $\equiv$   
 $x > 0 \wedge \exists c \in \{0..CROSSING\_COUNT - 1\} \bullet$   
 $(sendPoint1a[c] \leq x \vee sendPoint1b[c] \leq x \vee sendPoint2[c] \leq x)$   
 $\wedge vtpActive[RA.cr[c].vtpId] \wedge RA.vtp[RA.cr[c].vtpId].x \geq x$   
 $\wedge \neg closeRequested[c] \wedge \neg closeAcknowledged[c]$
- (C.4) **closeRequestNeededCrossings**  $\equiv$   
 $\{c \in \{0..CROSSING\_COUNT - 1\} \mid$   
 $(sendPoint1a[c] \leq x \vee sendPoint1b[c] \leq x \vee sendPoint2[c] \leq x)$   
 $\wedge vtpActive[RA.cr[c].vtpId] \wedge RA.vtp[RA.cr[c].vtpId].x \geq x$   
 $\wedge \neg closeRequested[c] \wedge \neg closeAcknowledged[c]\}$
- (C.5) **condResetCloseRequest**  $\equiv$   
 $\exists c \in \{0..CROSSING\_COUNT - 1\} \bullet$   
 $(RA.vtp[RA.cr[c].vtpId].x < x$   
 $\wedge (closeRequested[c] \vee closeAcknowledged[c]))$
- (C.6) **resetCloseRequestCrossings**  $\equiv$   
 $\{c \in \{0..CROSSING\_COUNT - 1\} \mid$   
 $RA.vtp[RA.cr[c].vtpId].x < x$   
 $\wedge (closeRequested[c] \vee closeAcknowledged[c])\}$
- (C.7) **condCloseRequestNoResponse**  $\equiv$   
 $\exists c \in \{0..CROSSING\_COUNT - 1\} \bullet$   
 $(closeRequested[c] \wedge \neg closeAcknowledged[c] \wedge$   
 $responseTimer[c] > GC.t_{reactResponse} + 2 \cdot GC.t_{transmit})$

$$(C.8) \text{ closeRequestNoResponseCrossings} \equiv \{c \in \{0..CROSSING\_COUNT - 1\} \mid \text{closeRequested}[c] \wedge \neg \text{closeAcknowledged}[c] \wedge \text{responseTimer}[c] > GC.t_{\text{reactResponse}} + 2 \cdot GC.t_{\text{transmit}}\}$$

$$(C.9) \text{ condCloseRequestTooManyTries} \equiv \text{repeatCounter}[cid] \geq GC.\text{repeatCount}_{max}$$

$$(C.10) \text{ algeClosePoint1a} \equiv \forall c \in \{0..CROSSING\_COUNT - 1\} \bullet \text{sendPoint1a}[c] := RA.vtp[RA.cr[c].vtpId].x + \frac{(c_v \cdot v)^2}{2 \cdot GC.a_{min}} - c_v \cdot v \cdot (4 \cdot GC.t_{\text{transmit}} + 2 \cdot GC.t_{\text{reactResponse}}) - 3 \cdot s_{\text{safe}}$$

$$(C.11) \text{ algeClosePoint1b} \equiv \forall c \in \{0..CROSSING\_COUNT - 1\} \bullet \text{sendPoint1}[c] := RA.vtp[RA.cr[c].vtpId].x + \frac{(c_v \cdot v)^2}{2 \cdot GC.a_{min}} - c_v \cdot v \cdot (3 \cdot GC.t_{\text{transmit}} + GC.t_{\text{reactResponse}} + GC.t_{\text{reactClose}} + RA.cr[c].t_{\text{yellow}} + RA.cr[c].t_{\text{gate}}) - 3 \cdot s_{\text{safe}}$$

$$(C.12) \text{ algeClosePoint2} \equiv \forall c \in \{0..CROSSING\_COUNT - 1\} \bullet \text{sendPoint2}[c] := RA.vtp[RA.cr[c].vtpId].x - c_v \cdot v \cdot (GC.t_{\text{transmit}} + GC.t_{\text{reactClose}} + RA.cr[c].t_{\text{approach}}) - 2 \cdot s_{\text{safe}}$$

**Close request status** Two variables store the close request status per crossing (initially *false* each):

**closeRequested** If a close request is sent to the respective railroad crossing, this variable is set to *true*. As soon as the closing is acknowledged, it is reset to *false*.

**closeAcknowledged** If the closing is acknowledged from the respective crossing, this variable is set to *true*. On reaching the crossing's velocity target point, it is reset to *false*.

**Close request protocol** A normal run of the communication protocol between train and crossing is:

1. The train sends a close request to the crossing:  $\text{closeRequested} \wedge \neg \text{closeAcknowledged}$
2. The crossing acknowledges to start closing:  $\neg \text{closeRequested} \wedge \text{closeAcknowledged}$
3. The train projects the time at which the crossing is closed and then sends a status request to the crossing.
4. The crossing responds with status *safe*.
5. The train reaches the crossing's velocity target point:  $\neg \text{closeRequested} \wedge \neg \text{closeAcknowledged}$ .

**Close request point calculation** The close request points on the track at which close requests should be sent to the respective railroad crossings are determined in concurrent ways (from which the smallest point, i.e. the one that is reached earliest, is significant):

*Optimistic undamped journey duration* A normal protocol run is assumed, therefore the following time durations are regarded:

$GC.t_{\text{transmit}}$  The transmission time duration of a radio telegram.

$GC.t_{\text{reactResponse}}$  The reaction time duration of the crossing before acknowledging.

$GC.t_{\text{reactClose}}$  The reaction time duration of the crossing before starting the closing activity.

$GC.t_{\text{yellow}}$  The yellow light duration of the crossing.

$GC.t_{\text{gate}}$  The gate closing duration of the crossing.

Between protocol step 1 and 2 the transmission time for one radio telegram is needed. Between 2 and 3, on the one hand the crossing responds via radio telegram and on the other hand it closes itself (starting with yellow lights, ending with closing the gates) – therefore the longer of both durations is relevant. Next, after step 3, the train sends the status request, which takes again the telegram transmission time. Finally, the crossing responds with a status telegram, again lasting the telegram transmission time. For each reaction of the crossing a reaction delay is assumed.

The close request distance of the train is calculated from this times' sum, assuming that the train constantly runs with its actual velocity. At the end of the protocol, the train's distance from the crossing has to be at least its braking distance,<sup>2</sup> such that the train does not need to brake; therefore the close request distance and the braking distance are subtracted from the velocity target point of the crossing. Additionally, a distance tolerance  $s_{\text{safe}}$  is subtracted.

This is calculated by (C.10)  $\text{algeClosePoint1a}$  and (C.11)  $\text{algeClosePoint1b}$ , respectively.

*Crossing freeing duration* The time duration from a close request to the finished freeing of the crossing consists of:

$GC.t_{\text{transmit}}$  See above.

$GC.t_{\text{reactClose}}$  See above.

$GC.t_{\text{approach}}$  This is the maximum time duration an approaching road user needs to either stop before the crossing or completely pass it.

Within the sum of these time durations the train is assumed to retain its (constant) velocity, thus a crossing freeing distance and further a respective crossing freeing location follows.

This is calculated by (C.12)  $\text{algeClosePoint2}$ .

---

<sup>2</sup>See mode `brakePointControl` for the calculation of the braking distance.

**Close request emission** The emission of a close request is guarded by (C.3) `condCloseRequestNeeded`:

The condition checks if the train has reached the send point for any crossing, and if a close request is needed for one of those. Alternatively, if all requests had been sent yet, the crossings had acknowledged closing, or the train had passed the crossing's velocity target points, the condition is not satisfied. If it is, it enables the respective transition from `waiting` and also forces the leaving of the mode. (C.4) `closeRequestNeededCrossings` non-deterministically chooses one of the send points.

**Close request receipt** As soon as the crossing acknowledges the closing by `crossingActivated(...)` while its velocity target point is in front of the train yet, the close request is finished and the acknowledged status is saved.

**Close request protocol failure** In order to detect protocol failures, a defined response time and a defined number of request tries are used by two conditions:

1. The condition (C.7) `condCloseRequestNoResponse` compares the response timer with the defined response time (that consists of the reaction time of the crossing and the transmission time for radio telegrams – request and acknowledge).  
If it evaluates to *true*, the respective transition is taken, one crossing is chosen by (C.8) `closeRequestNoResponseCrossings`, the close request status and timer are reset, and the request try counter is incremented.
2. The condition (C.9) `condCloseRequestTooManyTries` is checked whether the number of close request tries exceeds the defined maximum number.  
If it does, the crossing's failure is reported to the operations center. Note that the controller will continue sending close requests, however.

If the crossing explicitly rejects the close request by `crossingNotActivated(...)`, this is simply ignored – thus a protocol failure will be detected as if the crossing had not answered, and no closing of the crossing is assumed.

**Close request reset** The condition (C.5) `condResetCloseRequest` guards the reset of close requests: if a velocity target point of a crossing with pending or acknowledged close request is passed, the request status is completely reset since the crossing has been (or is being) passed and therefore will not be closed anymore (within the current lap on the train's circular track).

### C.2.11 CrossingStatusController

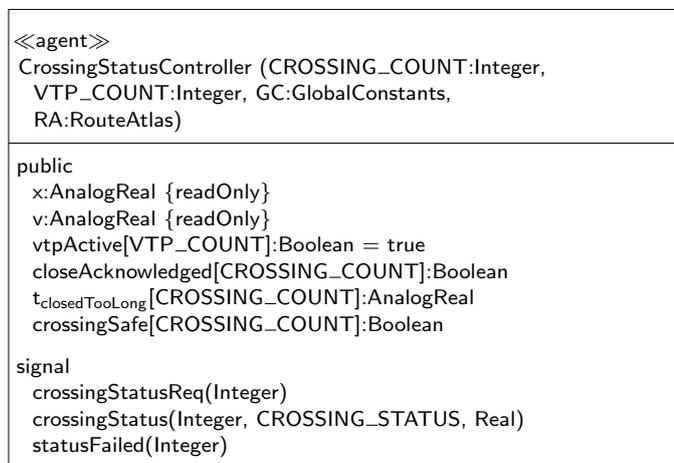


Figure C.22: Agent CrossingStatusController.

This agent defines the management of status requests that are sent to crossings that the train approaches:

In order to decide if a status request has to be sent, the controller needs information about the crossings (from the route atlas RA), as well as the current location  $x$  and velocity  $v$  of the train. A flag `closeAcknowledged` per crossing indicates if the crossing has acknowledged its closing<sup>3</sup>.

The main output of this agent is a flag `vtpActive` for each velocity target point that denotes if the target point is active or not – and therefore if the associated crossing is unsafe or safe. This is based on the variables `crossingSafe` which are calculated internally. Additionally, `crossingSafe` can be reset externally by `BrakePointController`.

Further, there is one timer `tclosedTooLong` for each crossing that represents the remaining safe closing time.<sup>4</sup> These timers are mainly controlled by `BrakePointController`, but are reset by this agent.

If a status request is needed, the signal `crossingStatusReq(...)` is sent. Possible results are received via `crossingStatus(...)`. `statusFailed(...)` is emitted, if the crossing's status can not be determined.

Several internal variables are used to store the crossing status, to store the request status, to check timeouts and to count the number of tries. A set of send points are continuously calculated for the determination of the request necessity.

<sup>3</sup>This is controlled by `CloseRequestController`.

<sup>4</sup>See `BrakePointController` or `brakePointControl`, respectively.

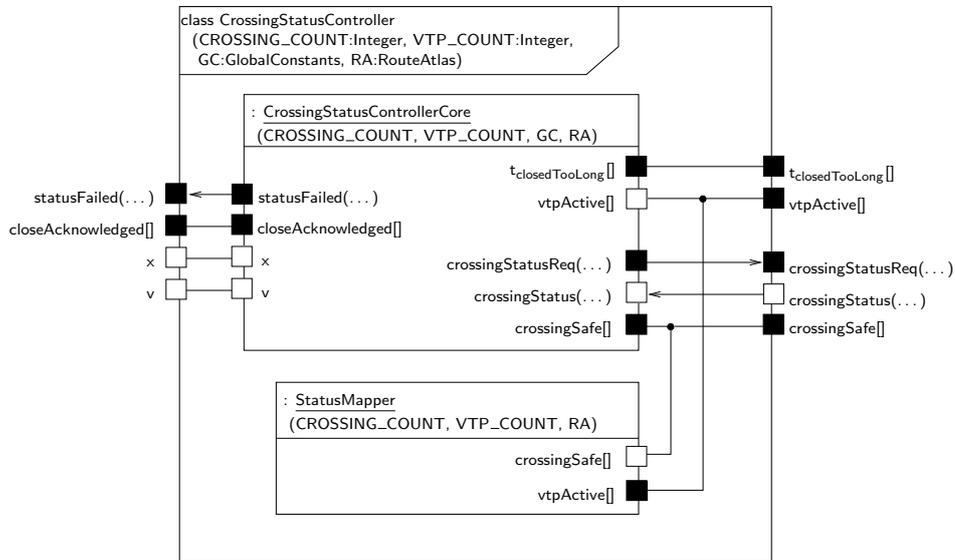


Figure C.23: Structure of CrossingStatusController.

The crossing status controller divides the main calculation of `crossingSafe` and the mapping between `crossingSafe` and `vtpActive` into sub-agents `CrossingStatusControllerCore` and `StatusMapper`. See the definitions of `CrossingStatusControllerCore` and `StatusMapper` for further explanations.

```

<<agent>>
CrossingStatusControllerCore (CROSSING_COUNT:Integer,
VTP_COUNT:Integer, GC:GlobalConstants,
RA:RouteAtlas)

public
x:AnalogReal {readOnly}
v:AnalogReal {readOnly}
vtpActive[VTP_COUNT]:Boolean {readOnly}
closeAcknowledged[CROSSING_COUNT]:Boolean
tclosedTooLong[CROSSING_COUNT]:AnalogReal
crossingSafe[CROSSING_COUNT]:Boolean = false

private
statusRequested[CROSSING_COUNT]:Boolean = false
responseTimer[CROSSING_COUNT]:AnalogReal = 0
repeatCounter[CROSSING_COUNT]:Integer = 0
sendPoint[CROSSING_COUNT]:AnalogReal = 0
cid:Integer = 0
time:Real = 0
status:CROSSING_STATUS = CROSSING_STATUS::UNKNOWN
cv:Real = GC.cv {readOnly}
ssafe:Real = GC.ssafe {readOnly}

signal
crossingStatusReq(Integer)
crossingStatus(Integer, CROSSING_STATUS, Real)
statusFailed(Integer)

crossingStatusControl

```

Figure C.24: Agent CrossingStatusControllerCore.

This agent defines the core part of the management of status requests that are sent to crossings that the train approaches. Its interface mainly corresponds with `CrossingStatusController`<sup>5</sup>.

This core controller determines the crossing state (`crossingSafe`) in consideration of `vtpActive`, thus clearly separating `crossingSafe` and `vtpActive`. But there is a direct interrelationship between them:  $\neg vtpActive \Leftrightarrow crossingSafe$  (for the respective crossing). This is ensured by the parallel agent `StatusMapper`.

The top-level mode that defines the behavior of this agent is `crossingStatusControl`.

<sup>5</sup>For interface details see the definition of `CrossingStatusController`.

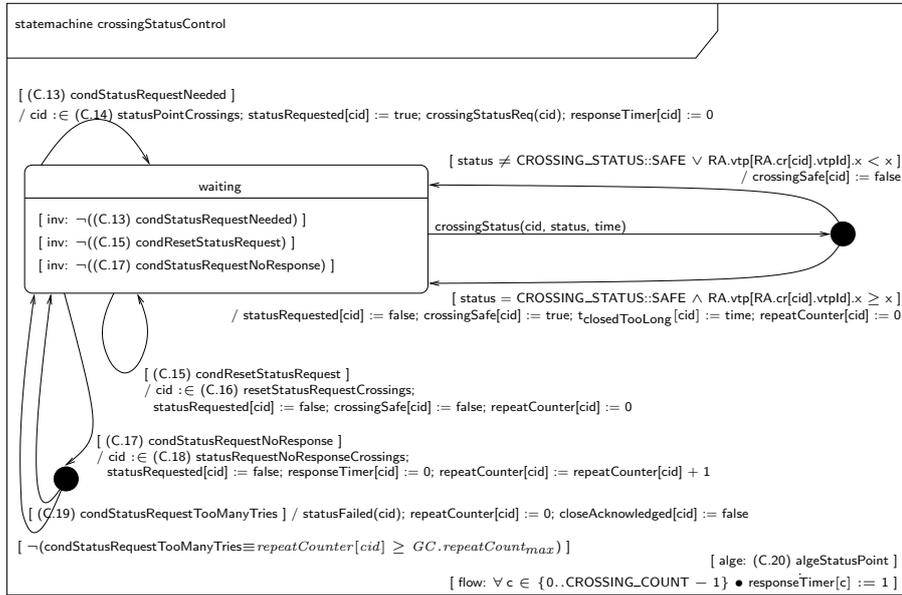


Figure C.25: Behavior of CrossingStatusControllerCore.

- (C.13) **condStatusRequestNeeded** ≡  
 $\exists c \in \{0..CROSSING\_COUNT - 1\} \bullet$   
 $(sendPoint[c] \leq x \wedge vtpActive[RA.cr[c].vtpId]$   
 $\wedge RA.vtp[RA.cr[c].vtpId].x \geq x \wedge \neg statusRequested[c]$   
 $\wedge \neg crossingSafe[c] \wedge closeAcknowledged[c])$
- (C.14) **statusPointCrossings** ≡  
 $\{c \in \{0..CROSSING\_COUNT - 1\} \mid$   
 $sendPoint[c] \leq x \wedge vtpActive[RA.cr[c].vtpId]$   
 $\wedge RA.vtp[RA.cr[c].vtpId].x \geq x \wedge \neg statusRequested[c]$   
 $\wedge \neg crossingSafe[c] \wedge closeAcknowledged[c]\}$
- (C.15) **condResetStatusRequest** ≡  
 $\exists c \in \{0..CROSSING\_COUNT - 1\} \bullet$   
 $(RA.cr[c].x_{end} < x \wedge (statusRequested[c] \vee crossingSafe[c]))$
- (C.16) **resetStatusRequestCrossings** ≡  
 $\{c \in \{0..CROSSING\_COUNT - 1\} \mid$   
 $RA.cr[c].x_{end} < x \wedge (statusRequested[c] \vee crossingSafe[c])\}$
- (C.17) **condStatusRequestNoResponse** ≡  
 $\exists c \in \{0..CROSSING\_COUNT - 1\} \bullet$   
 $(statusRequested[c] \wedge \neg crossingSafe[c] \wedge$   
 $responseTimer[c] > GC.t_{reactResponse} + 2 \cdot GC.t_{transmit})$
- (C.18) **statusRequestNoResponseCrossings** ≡  
 $\{c \in \{0..CROSSING\_COUNT - 1\} \mid$   
 $statusRequested[c] \wedge \neg crossingSafe[c] \wedge$   
 $responseTimer[c] > GC.t_{reactResponse} + 2 \cdot GC.t_{transmit}\}$

(C.19) **condStatusRequestTooManyTries**  $\equiv$   
 $repeatCounter[cid] \geq GC.repeatCount_{max}$

(C.20) **algeStatusPoint**  $\equiv$   
 $\forall c \in \{0..CROSSING\_COUNT - 1\} \bullet sendPoint[c] :=$   
 $RA.vtp[RA.cr[c].vtpId].x + \frac{(c_v \cdot v)^2}{2 \cdot GC.a_{min}} -$   
 $c_v \cdot v \cdot (2 \cdot GC.t_{transmit} + GC.t_{reactResponse}) - 2 \cdot s_{safe}$

**Status request status** Two variables store the status request status per crossing:

**closeAcknowledged** If the beginning of the closing activity is acknowledged from the respective crossing, this variable is *true*. This value is provided by agent `CloseRequestController`.

**statusRequested** If a status request is sent to the respective railroad crossing, this variable is set to *true*. As soon as the status is received, it is reset to *false*.

**Status request protocol** Status requests are part of the close request protocol: see section C.2.10 .

**Status request point calculation** The status request points on the track at which status requests should be sent to the respective railroad crossings are determined in the following way:<sup>6</sup>

A normal protocol run is assumed, therefore the following time durations are regarded:

$GC.t_{transmit}$  The transmission time duration of a radio telegram.

$GC.t_{reactResponse}$  The reaction time duration of the crossing before acknowledging.

The train sends the status request, which takes the telegram transmission time. The crossing responds with a status telegram, again lasting the telegram transmission time. For the reaction of the crossing a reaction delay is assumed.

The status request distance of the train is calculated from this times' sum, assuming that the train constantly runs with its actual velocity. At the end of the protocol, the train's distance from the crossing has to be at least its braking distance,<sup>7</sup> such that the train does not need to brake; therefore the status request distance and the braking distance are subtracted from the velocity target point of the crossing. Further, an additional distance tolerance  $s_{safe}$  is subtracted. This is calculated by (C.20) `algeStatusPoint`.

**Status request emission** The emission of a status request is guarded by the condition (C.13) `condStatusRequestNeeded`: The condition checks if the train has reached any send point (calculated from (C.20) `algeStatusPoint`) and whether a status request is needed, e.g. the velocity target point is active, the train has not passed it yet, the status is not requested yet, the crossing is not

<sup>6</sup>See also section C.2.10: Optimistic undamped journey duration

<sup>7</sup>See mode `brakePointControl` for the calculation of the braking distance.

safe yet, and the beginning of the closing of the crossing is acknowledged. If the condition evaluates to *true*, it enables the respective transition from *waiting* and also forces the leaving of the mode. (C.14) *statusPointCrossings* non-deterministically chooses one of the affected crossings, and the status request is sent.

**Status request reception** As soon as the crossing acknowledges the *safe* status by *crossingStatus(...)* while its velocity target point is in front of the train yet, the status request is finished and the crossing status is saved. The crossing’s safe close duration timer is set according to the status message.

**Status request protocol failure** In order to detect protocol failures, a defined response time and a defined number of request tries are used by two conditions:

1. The condition (C.17) *condStatusRequestNoResponse* compares the response timer with the defined response time (that consists of the reaction time of the crossing and the transmission time for radio telegrams – request and status message).  
If it evaluates to *true*, the respective transition is taken, one crossing is chosen by (C.18) *statusRequestNoResponseCrossings*, the status request status and timer are reset, and the request try counter is incremented.
2. The condition (C.19) *condStatusRequestTooManyTries* is checked if the number of status request tries exceeds the defined maximum number. If it does, the crossing’s failure is reported to the operations center. The controller will nevertheless continue sending status requests.

If the crossing explicitly states a different status than *safe*, this is simply ignored – thus a protocol failure will be detected as if the crossing had not answered, and an unsafe status of the crossing is assumed.

**Status request reset** The condition (C.15) *condResetStatusRequest* guards the reset of status requests: if a velocity target point of a crossing with pending or confirmed status is passed, the request status is completely reset since the crossing has been (or is being) passed and therefore will not be closed anymore (within the current lap on the train’s circular track).

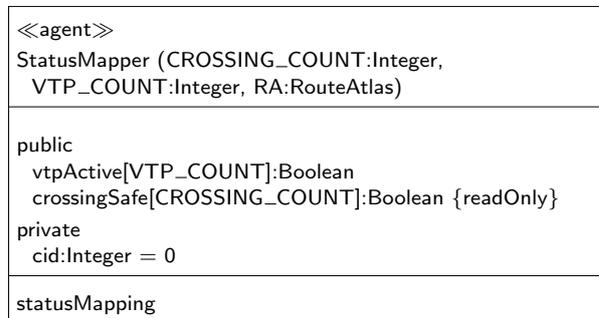


Figure C.26: Agent StatusMapper.

This agent defines an auxiliary part of the management of status requests that are sent to crossings that the train approaches:

The controller `CrossingStatusControllerCore` determines the crossing state (`crossingSafe`) in consideration of `vtpActive`, thus clearly separating `crossingSafe` and `vtpActive`. But there is a direct interrelationship between them:  $\neg vtpActive \Leftrightarrow crossingSafe$  (for the respective crossing).

To ensure this is the task of this agent.

Additional information to `vtpActive` and `crossingSafe` is provided through the route atlas `RA` for the mapping between crossings and velocity target points.

The top-level mode that defines the behavior of this agent is `statusMapping`.

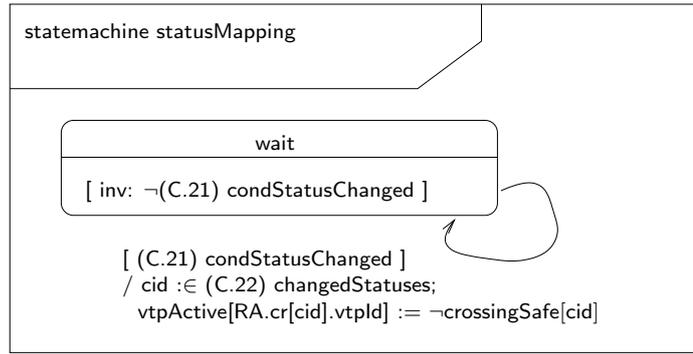


Figure C.27: Behavior of StatusMapper.

(C.21) **condStatusChanged**  $\equiv$   
 $\exists c \in \{0..CROSSING\_COUNT - 1\} \bullet$   
 $crossingSafe[c] \neq \neg vtpActive[RA.cr[c].vtpId]$

(C.22) **changedStatuses**  $\equiv$   
 $\{c \in \{0..CROSSING\_COUNT - 1\} \mid$   
 $crossingSafe[c] \neq \neg vtpActive[RA.cr[c].vtpId]\}$

This statemachine establishes the interrelationship:  $\neg vtpActive \Leftrightarrow crossingSafe$  (for each crossing).

The condition (C.21) `condStatusChanged` enables and enforces the single transition, iff  $\neg crossingSafe$  and `vtpActive` differ for any crossing. Then, one of the affected crossings is chosen non-deterministically from (C.22) `changedStatuses`, and for this crossing the value of `vtpActive` is set to  $\neg crossingSafe$ .

## C.2.12 TrainRadioController

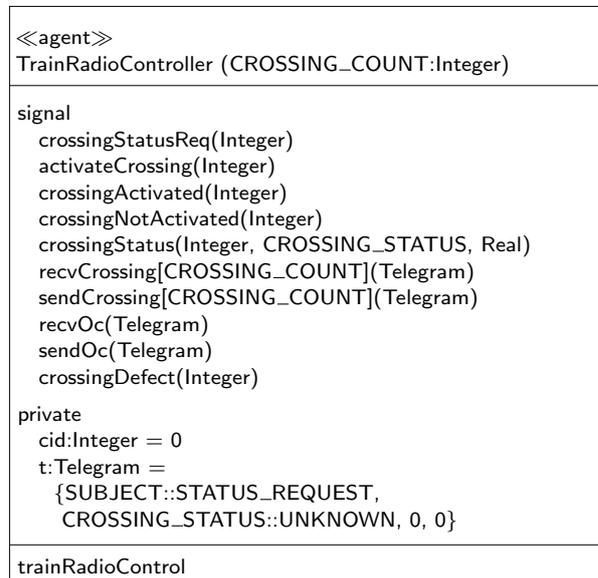


Figure C.28: Agent TrainRadioController.

The statemachine TrainRadioController maps the signals `crossingStatusReq(...)`, `activateCrossing(...)` and `crossingDefect(...)` to respective radio telegrams via `send...`, and maps incoming radio telegrams via `rcv...` to the signals `crossingActivated(...)`, `crossingNotActivated(...)`, and `crossingStatus(...)`. This is defined in top-level mode `trainRadioControl`.

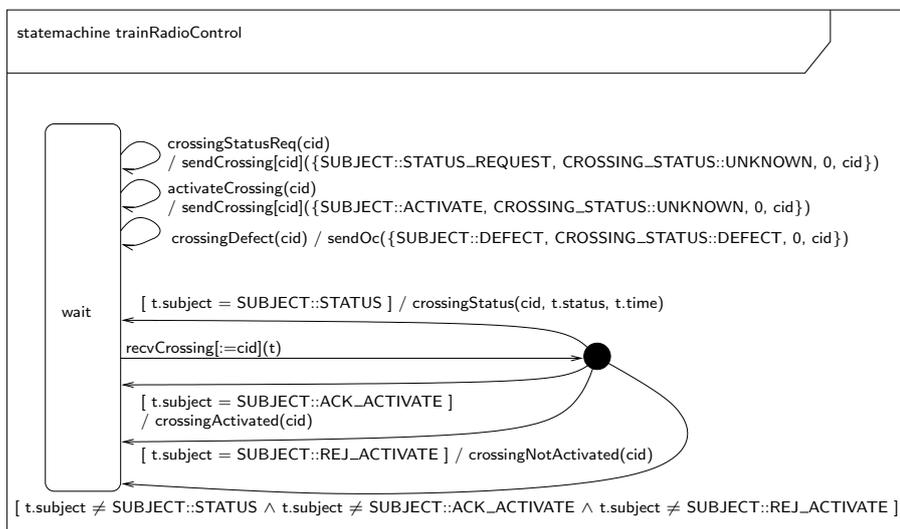


Figure C.29: Behavior of TrainRadioController.

This statemachine maps status messages and requests to the respective radio messages. See also section C.2.29: `crossingRadioControl`.

### C.2.13 UserInteractionController

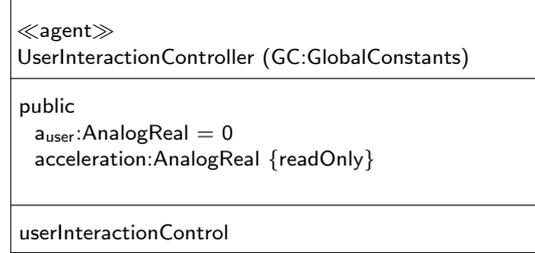


Figure C.30: Agent UserInteractionController.

This agent's purpose is to adjust the incoming user-requested acceleration within defined bounds (from GC). The resulting  $a_{\text{user}}$  is provided. The top-level mode `userInteractionControl` defines this behavior.

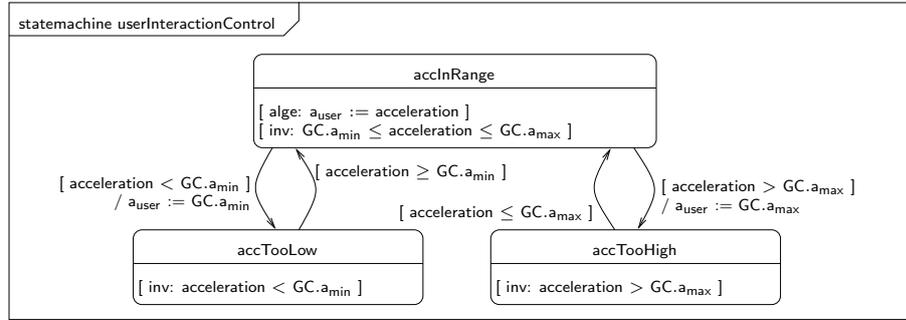


Figure C.31: Behavior of UserInteractionController.

While the user-provided acceleration is within the possible bounds, it is copied to  $a_{\text{user}}$  (mode `acInRange`). If one of the bounds is violated, `accTooLow` or `accTooHigh`, resp. is entered and the appropriate extreme value is used instead.

### C.2.14 LocalizationController

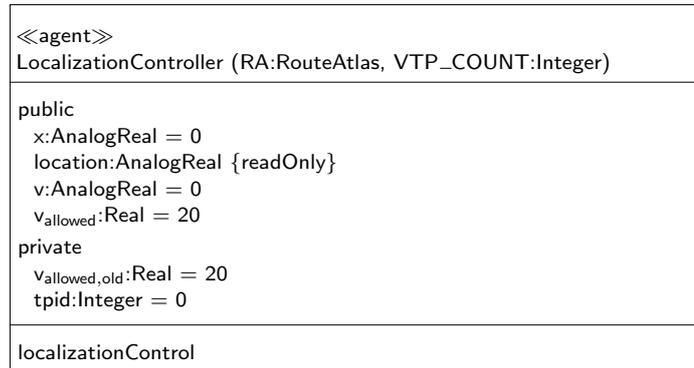


Figure C.32: Agent LocalizationController.

The localization agent of the train controller provides:

*Location on the track*  $x$  is the train's location on the track, relative to the track's beginning. The train's location information is provided by the controller's environment (e.g. by a location sensor) and thus is read as variable `location`.

*Current speed*  $v$  is the train's current velocity. This is the first derivative of  $x$  with respect to time.

*Currently allowed speed*  $v_{\text{allowed}}$  is the allowed (maximum) velocity for the train's current location. This depends on the information provided by the route atlas `RA`.

A copy is stored in  $v_{\text{allowed,old}}$ , because the agent `MovementController` changes the allowed speed in certain situations. The localization controller thus may restore it later.

The current speed and the allowed speed are provided by the localization controller, because they directly depend on its location.

The top-level mode `localizationControl` defines the behavior.

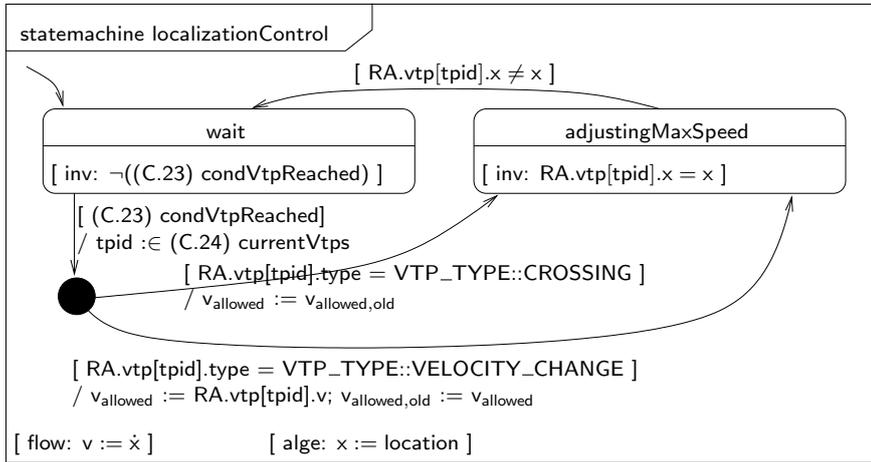


Figure C.33: Behavior of LocalizationController.

(C.23) **condVtpReached**  $\equiv$   
 $\exists i \in \{0..VTP\_COUNT - 1\} \bullet RA.vtp[i].x = x$

(C.24) **currentVtps**  $\equiv$   
 $\{i \in \{0..VTP\_COUNT - 1\} \mid RA.vtp[i].x = x\}$

The three provided outputs are calculated as follows:

$x$  This is simply always a copy of the incoming location `location`.

$v$  The velocity is determined by a flow constraint that defines it as the first derivative of  $x$  with respect to time.

$v_{\text{allowed}}$  The allowed velocity is set to the target velocity of the respective velocity target point whenever the train reaches one (that provides a new allowed velocity). If in contrast the train passes a crossing, then the previous allowed velocity is restored from  $v_{\text{allowed,old}}$ . This both is guarded by the constraint (C.23) `condVtpReached`.

## C.2.15 MovementController

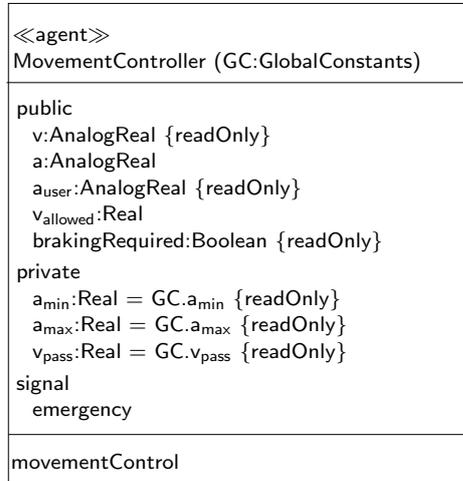


Figure C.34: Agent MovementController.

The movement controller determines the target acceleration  $\mathbf{a}$  that must be realized by the train's engine and brake. Therefore it reads the user-requested acceleration  $\mathbf{a}_{\text{user}}$  and either directly uses it, or adjusts it according to these constraints:

*Maximum allowed velocity* The currently allowed velocity  $v_{\text{allowed}}$  is used as upper bound. It is assumed that it is sufficient to consider the allowed velocity for the head of the train, and that (increasing) velocity changes are suitably located. Basically, **MovementController** reads the allowed velocity, but after enforced braking, it reduces the allowed velocity in order to avoid an immediate successive velocity increase.

*Velocity target point approach* The input variable **brakingRequired** denotes if the train must reduce its velocity according to the dynamic velocity profile (i.e. while it is approaching an active velocity target point).

*Emergency situation* If the signal **emergency** is received, then the train stops as soon as possible.

For these adjustments several constants as well as the current velocity  $\mathbf{v}$  are read.

The behavior is defined in **movementControl**.

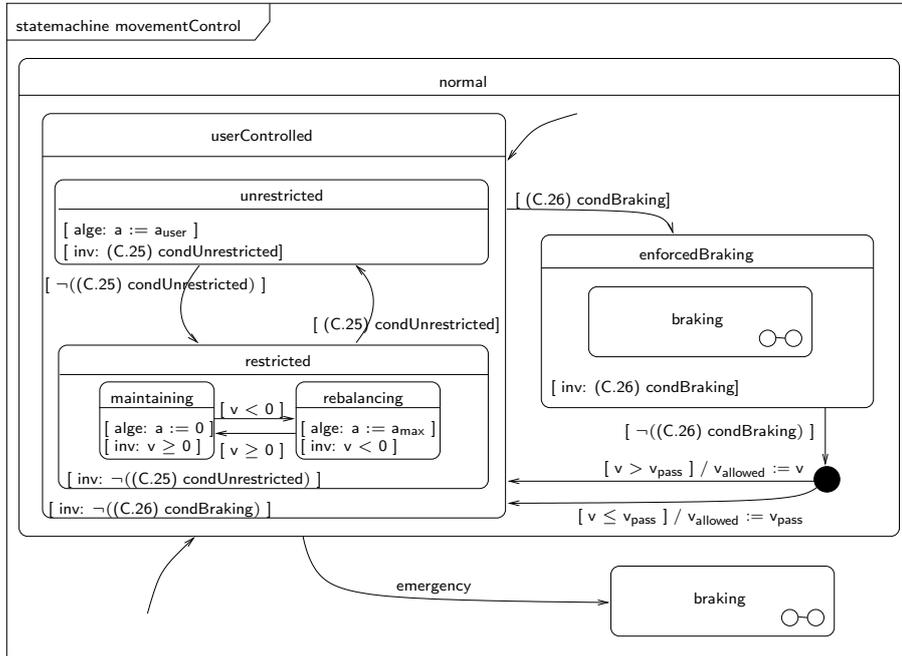


Figure C.35: Behavior of MovementController.

$$(C.25) \text{ condUnrestricted} \equiv (v < v_{allowed} \vee a_{user} < 0) \wedge (v > 0 \vee a_{user} > 0)$$

$$(C.26) \text{ condBraking} \equiv \text{brakingRequired} \vee v > v_{allowed}$$

This statemachine is divided into several submodes:

**normal** This mode subsumes normal movement.

**userControlled** The user controls the target acceleration.

**unrestricted** The user-requested acceleration is directly mapped to the target acceleration, because no constraint is violated.

**restricted** The restricted mode is entered, if the user tries to exceed the maximum allowed velocity or to fall below the speed 0. Otherwise, **unrestricted** is re-entered. Within, possible inaccuracies leading to velocities below 0 (i.e. backward train movement) are balanced.

**enforcedBraking** The train brakes automatically, therefore the user-required acceleration is discarded. This mode is entered if the actual velocity is higher than the maximum allowed velocity, or if braking is required because of the dynamic velocity profile. Otherwise, the mode **userControlled** is re-entered.

**braking** Within this mode, the maximum deceleration is applied as long as the train has not stopped yet (**stopping**), afterwards the train remains stopped (**stopped**) – thus a full braking is performed. Similar to mode **restricted**, inaccuracies are balanced.

**braking** This mode represents the emergency behavior; it is entered if the emergency signal is received. Its operation is exactly the same as **braking** within **enforcedBraking**, but there is no way to leave the mode, i.e. after having stopped the train will remain stopped.

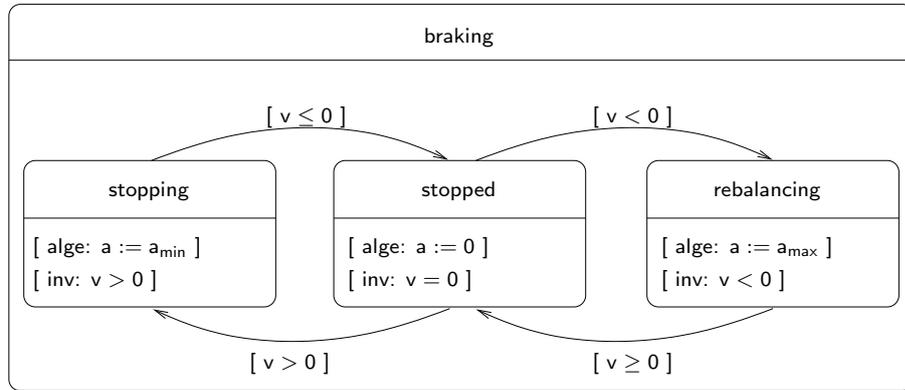


Figure C.36: Submode braking.

See movementControl.

### C.2.16 EmergencyController

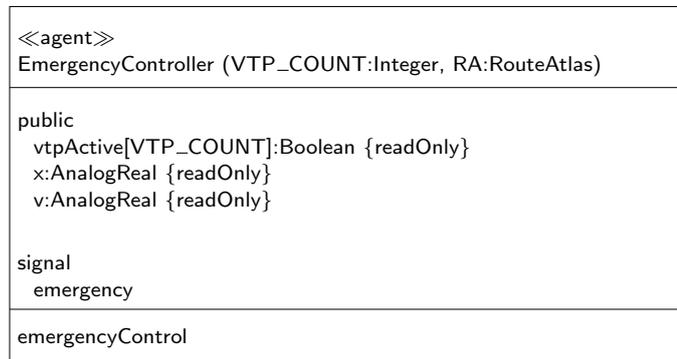


Figure C.37: Agent EmergencyController.

This agent detects emergency situations: If a velocity target point is reached while the train’s velocity  $v$  is higher than the target velocity,<sup>8</sup> an emergency signal is raised.

The train’s velocity  $v$ , its current location  $x$ , the velocity target points (listed in route atlas  $RA$ ) and their activity states are needed to determine this.

The behavior is defined in `emergencyControl`.

<sup>8</sup>A special case is the passing of the velocity target point of an unsafe crossing.

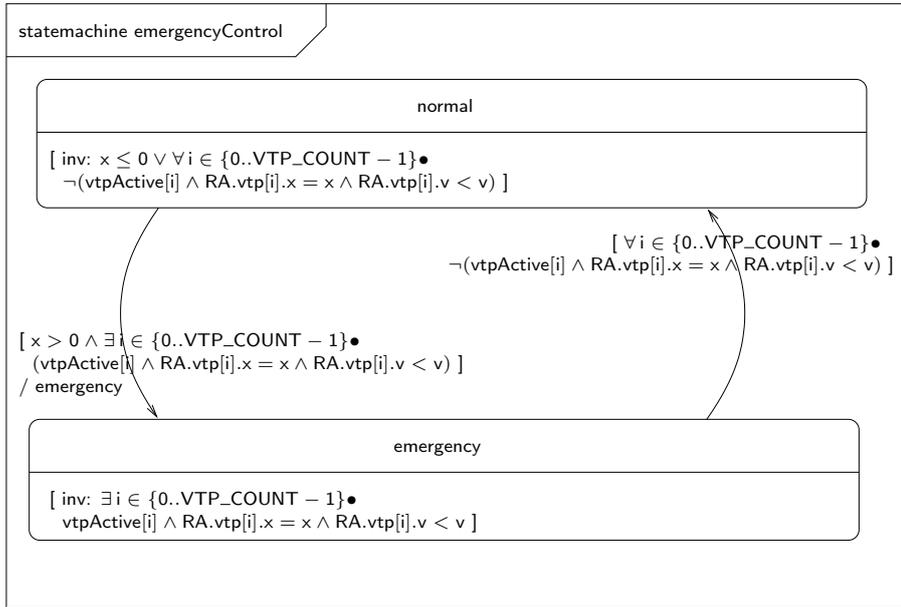


Figure C.38: Behavior of EmergencyController.

The distinction between normal and emergency situations is defined as condition: iff the train's location is an active velocity target point and the train is too fast (according to the target velocity of that target point), then an emergency situation is detected. This condition enables and enforces the transition to mode **emergency** and an **emergency** signal is raised. As soon as the condition does not hold, its negation is used to model the immediate mode change to mode **normal**.

## C.2.17 BrakePointController

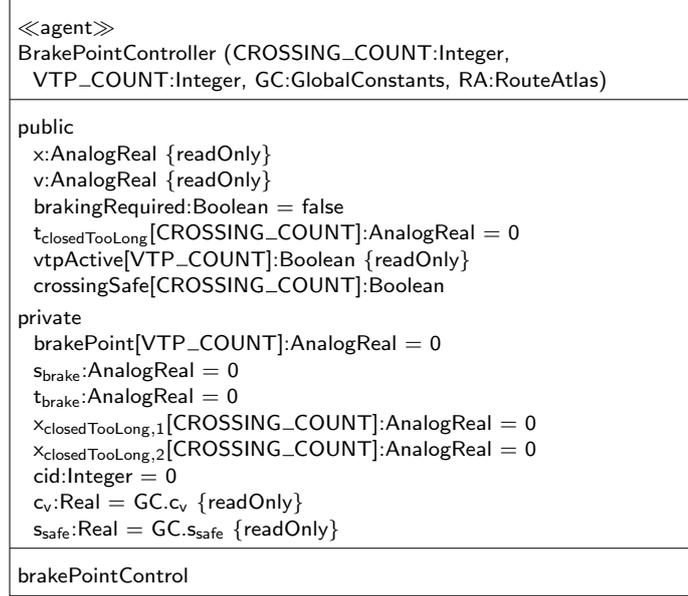


Figure C.39: Agent BrakePointController.

The brake point controller continuously determines, if the train must be braking because of any active velocity target point: `brakingRequired`. Therefore it calculates all brake points `brakePoint`: for each velocity target point, there is a location on the track at which the train must start to brake in order to reach the target velocity at the point.

The brake points depend on the current velocity  $v$  of the train, and on the locations of the target points that are contained in the route atlas `RA`.

Dynamic target point information is regarded: `vtpActive` denotes if the corresponding velocity target point is active or not, `tclosedTooLong` contains for each safe crossing the remaining safe time duration.

Two estimated train positions  $x_{\text{closedTooLong},1}$  and  $x_{\text{closedTooLong},2}$  for time `tclosedTooLong` are extrapolated in order to determine if the train will (probably) pass the crossing in time. It is assumed that anytime the train may stop within the braking distance  $s_{\text{brake}}$  and within the braking time  $t_{\text{brake}}$ .  $x_{\text{closedTooLong},1}$  is meaningful if `tclosedTooLong`  $\leq t_{\text{brake}}$ . Otherwise  $x_{\text{closedTooLong},2}$  is taken into account; its calculation includes the assumption, that the train will not completely stop but will cover the remaining distance apart of  $s_{\text{brake}}$  with the constant velocity  $\text{const.v}_{\text{pass}} > 0$ . The greater  $\text{const.v}_{\text{pass}}$  is, the more likely is the violation of the maximum safe time duration of the crossing, but the probability of unnecessary train stops at safe crossings decreases.

Provided that the train is too fast, `brakingRequired` is set, if the location  $x$  of the train has reached any brake point but has not passed the associated velocity target point which is active or belongs to a crossing that presumably will not be passed within the remaining safe time duration.

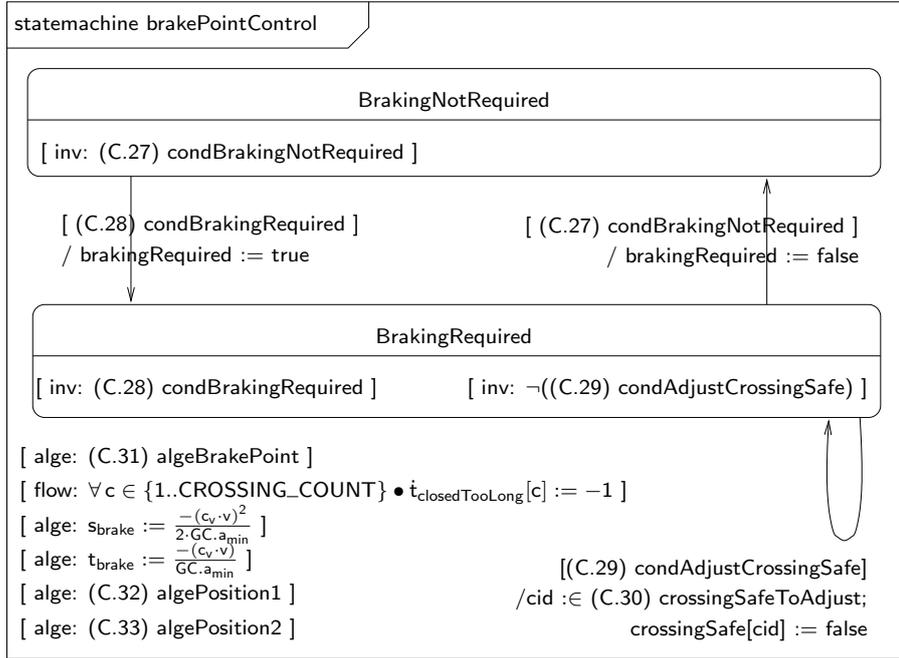


Figure C.40: Behavior of BrakePointController.

(C.27) **condBrakingNotRequired**  $\equiv$   
 $\forall i \in \{0..VTP\_COUNT - 1\} \bullet$   
 $\neg(brakePoint[i] \leq x \wedge RA.vtp[i].v < v \wedge RA.vtp[i].x > x \wedge$   
 $(vtpActive[i] \vee$   
 $RA.vtp[i].type = VTP\_TYPE::CROSSING \wedge$   
 $((x_{closedTooLong,1}[RA.vtp[i].crId]$   
 $\leq RA.cr[RA.vtp[i].crId].x_{end} + GC.trainLength$   
 $\wedge t_{closedTooLong}[RA.vtp[i].crId] \leq t_{brake})$   
 $\vee (x_{closedTooLong,2}[RA.vtp[i].crId]$   
 $\leq RA.cr[RA.vtp[i].crId].x_{end} + GC.trainLength$   
 $\wedge t_{closedTooLong}[RA.vtp[i].crId] > t_{brake})))$

(C.28) **condBrakingRequired**  $\equiv$   
 $\exists i \in \{0..VTP\_COUNT - 1\} \bullet$   
 $(brakePoint[i] \leq x \wedge RA.vtp[i].v < v \wedge RA.vtp[i].x > x \wedge$   
 $(vtpActive[i] \vee$   
 $RA.vtp[i].type = VTP\_TYPE::CROSSING \wedge$   
 $((x_{closedTooLong,1}[RA.vtp[i].crId]$   
 $\leq RA.cr[RA.vtp[i].crId].x_{end} + GC.trainLength$   
 $\wedge t_{closedTooLong}[RA.vtp[i].crId] \leq t_{brake})$   
 $\vee (x_{closedTooLong,2}[RA.vtp[i].crId]$   
 $\leq RA.cr[RA.vtp[i].crId].x_{end} + GC.trainLength$   
 $\wedge t_{closedTooLong}[RA.vtp[i].crId] > t_{brake})))$

$$\begin{aligned}
\text{(C.29) } \mathbf{condAdjustCrossingSafe} &\equiv \\
&\exists c \in \{0..CROSSING\_COUNT - 1\} \bullet \\
&\quad (x_{closedTooLong,1}[c] \leq RA.cr[c].x_{end} + GC.trainLength \\
&\quad \wedge t_{closedTooLong}[c] \leq t_{brake} \\
&\quad \vee x_{closedTooLong,2}[c] \leq RA.cr[c].x_{end} + GC.trainLength \\
&\quad \wedge t_{closedTooLong}[c] > t_{brake})
\end{aligned}$$

$$\begin{aligned}
\text{(C.30) } \mathbf{crossingSafeToAdjust} &\equiv \\
&\{c \in \{0..CROSSING\_COUNT - 1\} \mid \\
&\quad x_{closedTooLong,1}[c] \leq RA.cr[c].x_{end} + GC.trainLength \\
&\quad \wedge t_{closedTooLong}[c] \leq t_{brake} \\
&\quad \vee x_{closedTooLong,2}[c] \leq RA.cr[c].x_{end} + GC.trainLength \\
&\quad \wedge t_{closedTooLong}[c] > t_{brake}\}
\end{aligned}$$

$$\begin{aligned}
\text{(C.31) } \mathbf{algeBrakePoint} &\equiv \\
&\forall i \in \{0..VTP\_COUNT - 1\} \bullet \\
&\quad brakePoint[i] := RA.vtp[i].x - \frac{RA.vtp[i].v^2 - (c_v \cdot v)^2}{2 \cdot GC.a_{min}} - s_{safe}
\end{aligned}$$

$$\begin{aligned}
\text{(C.32) } \mathbf{algePosition1} &\equiv \\
&\forall c \in \{0..CROSSING\_COUNT - 1\} \bullet \\
&\quad x_{closedTooLong,1}[c] := x + \frac{GC.a_{min}}{2} \cdot t_{closedTooLong}[c]^2 \\
&\quad + c_v \cdot v \cdot t_{closedTooLong}[c] - s_{safe}
\end{aligned}$$

$$\begin{aligned}
\text{(C.33) } \mathbf{algePosition2} &\equiv \\
&\forall c \in \{0..CROSSING\_COUNT - 1\} \bullet \\
&\quad x_{closedTooLong,2}[c] := x + s_{brake} + (t_{closedTooLong}[c] - t_{brake}) \cdot GC.v_{pass} \\
&\quad - s_{safe}
\end{aligned}$$

The statemachine `brakePointControl` has two submodes that correspond to the value of the output variable `brakingRequired`. There is one condition that decides which is the active mode and thus if `brakingRequired` is true or false – `condBrakingRequired`.<sup>9</sup> It is applied as combination of invariants of the modes and guards of the transition to enforce the immediate mode change as soon as its value changes. Braking is required, iff there is any velocity target point, such that

1. the train has reached the brake point of the target point, and
2. the actual velocity of the train is greater than the target velocity, and
3. the train has not reached the target point yet, and
4. (a) either the target point is active, or  
(b) the target point is associated with a railroad crossing that probably will not be passed within the remaining safe closing duration `tclosedTooLong` by the train.

---

<sup>9</sup>`condBrakingNotRequired` is simply the negation of `condBrakingRequired`.

**Brake point calculation** For the calculation of the brake points a constant (negative) acceleration  $GC.a_{min}$  is assumed:

$$a \equiv a_{brake}(t) = GC.a_{min}$$

Therefore, the velocity decreases linearly over time  $t$ , starting with velocity  $v_0$ :

$$v_{brake}(t) = a \cdot t + v_0$$

The train position is the integral of the velocity:

$$x_{brake}(t) = \frac{a}{2} \cdot t^2 + v_0 \cdot t + x_0$$

With target velocity  $v_{target}$  the time duration  $t_{target}$  results:

$$v_{brake}(t_{target}) = a \cdot t_{target} + v_0 = v_{target} \Rightarrow t_{target} = \frac{v_{target} - v_0}{a}$$

The target position then is:

$$\begin{aligned} & x_{brake}(t_{target}) \\ &= \frac{a}{2} \cdot t_{target}^2 + v_0 \cdot t_{target} + x_0 \\ &= \frac{a}{2} \cdot \left( \frac{v_{target} - v_0}{a} \right)^2 + v_0 \cdot \frac{v_{target} - v_0}{a} + x_0 \\ &= \frac{a}{2} \cdot \frac{(v_{target} - v_0)^2}{a^2} + v_0 \cdot \frac{v_{target} - v_0}{a} + x_0 \\ &= \frac{(v_{target} - v_0)^2}{2a} + \frac{v_0 \cdot (v_{target} - v_0)}{a} + x_0 \\ &= \frac{(v_{target} - v_0)^2}{2a} + \frac{2 \cdot v_0 \cdot (v_{target} - v_0)}{2a} + x_0 \\ &= \frac{(v_{target} - v_0)^2 + 2 \cdot v_0 \cdot (v_{target} - v_0)}{2a} + x_0 \\ &= \frac{v_{target}^2 - 2 \cdot v_{target} \cdot v_0 + v_0^2 + 2 \cdot v_0 \cdot v_{target} - 2v_0^2}{2a} + x_0 \\ &= \frac{v_{target}^2 - v_0^2}{2a} + x_0 \end{aligned}$$

The braking distance is the difference between target position and current train position:

$$s_{target} \equiv s_{brake}(t_{target}) = x_{brake}(t_{target}) - x_0 = \frac{v_{target}^2 - v_0^2}{2a}$$

Thus the brake point associated with a velocity target point  $x_{target}$  is

$$x_{bp} = x_{target} - s_{target}$$

and with  $v \equiv v_0$ ,  $ra.vtp[i].v \equiv v_{target}$ ,  $ra.vtp[i].x \equiv x_{target}$ ,  $brakePoint[i] \equiv x_{bp}$  for each velocity target point  $i$  follows (C.31) `algeBrakePoint`. There are two additional parameters which are used to model absolute and relative safety tolerances: coefficient  $c_v$  is multiplied with the velocity, and distance offset  $s_{safe}$  simply is subtracted.

**Safe closing time duration timer** The remaining safe closing time of the crossings is modeled as decreasing clocks for each crossing.

**Braking distance and time** The definition of  $s_{\text{brake}}$  and  $t_{\text{brake}}$  is a special case of  $s_{\text{brake}}(t_{\text{target}})$  and  $t_{\text{target}}$ , resp. with  $v_{\text{target}} = 0$ .

**Extrapolated train location at  $t_{\text{closedTooLong}}$**  The values  $x_{\text{closedTooLong},1}$  and  $x_{\text{closedTooLong},2}$  are continuously updated as defined in the algebraic conditions in order to model the discontinuous function

$$x_{\text{closedTooLong}} = \begin{cases} x_{\text{closedTooLong},1}; & t_{\text{closedTooLong}}[\text{ra.vtp}[i].\text{cr.id}] \leq t_{\text{brake}} \\ x_{\text{closedTooLong},2}; & t_{\text{closedTooLong}}[\text{ra.vtp}[i].\text{cr.id}] > t_{\text{brake}} \end{cases}$$

since algebraic conditions must define continuous functions. Therefore, the case differentiation is modeled within `condBrakingRequired`.

$x_{\text{closedTooLong},1}$  is calculated with the assumption that the train brakes according to  $x_{\text{brake}}(t)$ , thus from

$$\begin{aligned} x_{\text{extra},1}(t_{\text{closedTooLong}}) &= x_{\text{brake}}(t_{\text{closedTooLong}}) \\ &= \frac{a}{2} \cdot t_{\text{closedTooLong}}^2 + v_0 \cdot t_{\text{closedTooLong}} + x_0 \end{aligned}$$

with  $x_{\text{closedTooLong},1}[\text{c}] \equiv x_{\text{extra},1}(t_{\text{closedTooLong}})$ ,  $v \equiv v_0$ ,  $t_{\text{closedTooLong}}[\text{c}] \equiv t_{\text{closedTooLong}}$ ,  $x \equiv x_0$  for each crossing  $\text{c}$  follows (C.32) `algePosition1`.

For  $x_{\text{closedTooLong},2}$  it is assumed that  $t_{\text{closedTooLong}} > t_{\text{brake}}$ , such that for the time duration  $t_{\text{brake}}$  the train brakes and therefore covers the distance  $s_{\text{brake}}$ , while for the remaining time  $t_{\text{pass}} = t_{\text{closedTooLong}} - t_{\text{brake}}$  it has a constant velocity  $v_{\text{pass}}$  and therefore covers the distance  $s_{\text{pass}} = v_{\text{pass}} \cdot t_{\text{pass}}$ :

$$\begin{aligned} x_{\text{extra},2}(t_{\text{closedTooLong}}) &= s_{\text{brake}} + s_{\text{pass}} + x_0 \\ &= s_{\text{brake}} + v_{\text{pass}} \cdot t_{\text{pass}} + x_0 \\ &= s_{\text{brake}} + v_{\text{pass}} \cdot (t_{\text{closedTooLong}} - t_{\text{brake}}) + x_0 \end{aligned}$$

With  $x_{\text{closedTooLong},2}[\text{c}] \equiv x_{\text{extra},2}(t_{\text{closedTooLong}})$ ,  $\text{GC.v}_{\text{pass}} \equiv v_{\text{pass}}$ ,  $t_{\text{closedTooLong}}[\text{c}] \equiv t_{\text{closedTooLong}}$ ,  $x \equiv x_0$  for each crossing  $\text{c}$  follows (C.33) `algePosition2`.

## C.2.18 Crossing

<pre> &lt;&lt;agent&gt;&gt; Crossing (GC:GlobalConstants, RA:RouteAtlas)  public   trainLocation:AnalogReal   occupied:Boolean  signal   recvTrain(Telegram)   sendTrain(Telegram)   recvOc(Telegram)   sendOc(Telegram)   repairCrossing </pre>
--

Figure C.41: Agent Crossing.

The agent **Crossing** represents complete crossings which are composed of gates, lights etc. The components are defined in the composite structure diagram.

The crossing is embedded into the overall system, where it communicates via radio channels with the train and the operations center. Therefore, the signal specifications are provided. Further, the crossing can be repaired on the initiative of the operations center.

The status of the crossing's danger zone, as well as the train's length and actual position, are also part of the interface, because they are shared with the operations center or the train, respectively.

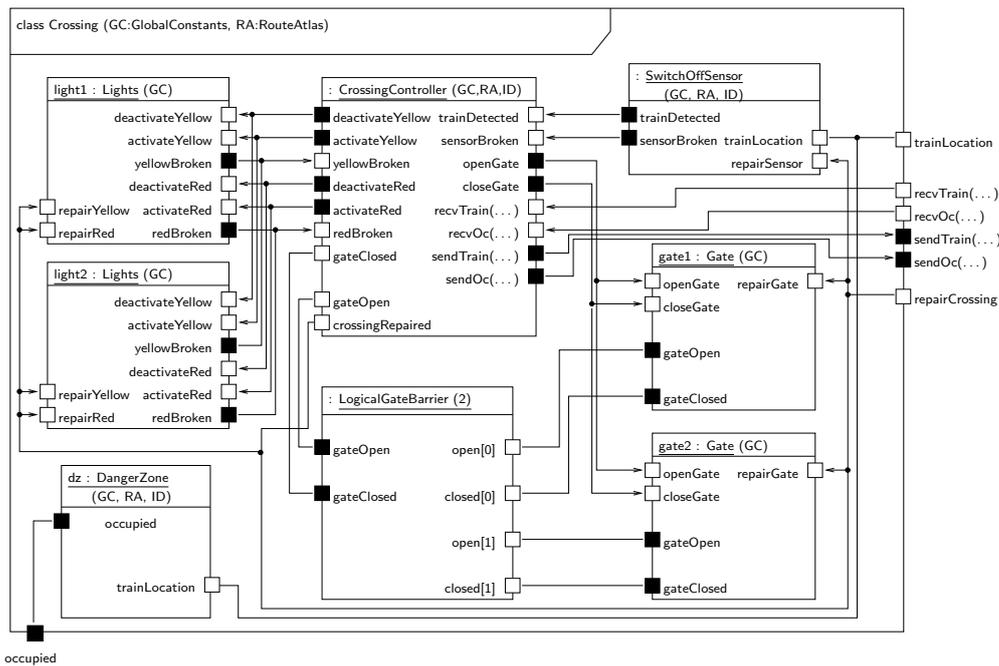


Figure C.42: Structure of Crossing.

The crossing is composed of:

*Danger zone* The critical area where cars, pedestrians etc. cross the tracks.

*Lights* Two sets of traffic lights, each consisting of a red and a yellow light.

*Crossing controller* The controlling hardware/software of the crossing.

*LogicalGateBarrier* A logical aggregation that provides a single opening status of all barriers.

*Gate* Two gates consisting of a barrier and a motor each.

*Switch-off sensor* The sensor that detects the train when it has passed the crossing. In this case, the crossing can be switched off.

The crossing structure diagram defines the connections between the crossing components and the crossing controller: The gates are directly triggered by the

respective signals from the crossing controller, whereas the logical gate barrier abstracts away from the quantity of gates and provides the crossing controller a single opening status. The lights are also directly triggered by the controller.

The switch-off sensor notifies the controller if the train passes, whereas the danger zone provides its occupancy status directly to the environment. Both get the train's location and length.

The controller communicates with the environment by means of radio telegrams that are received from or send to the train and the operations center.

The complete crossing may be repaired, thus all components are repaired.

### C.2.19 Lights



Figure C.43: Agent Lights.

A pair of lights (yellow and red) behaves like two single lights, see agent Light.

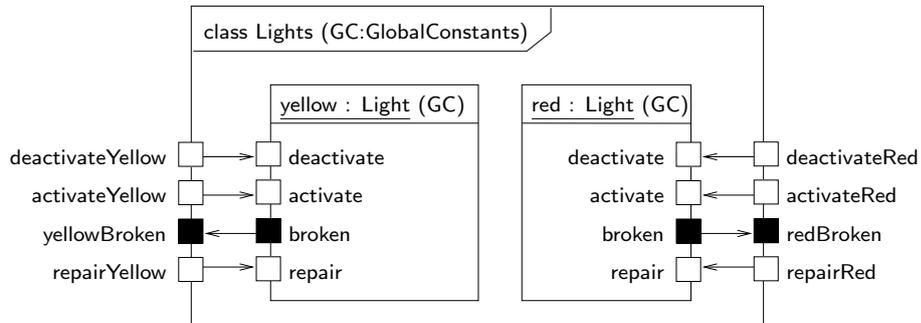


Figure C.44: Structure of Lights.

The signals of the pair of lights are mapped to the corresponding signals of the contained lights.

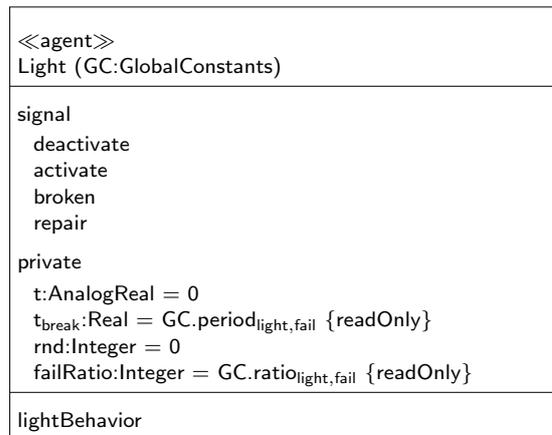


Figure C.45: Agent Light.

A single light can be activated or deactivated. Furthermore, the signal broken is sent on light defect. It can be repaired again.

The light behavior is defined as top-level mode lightBehavior.

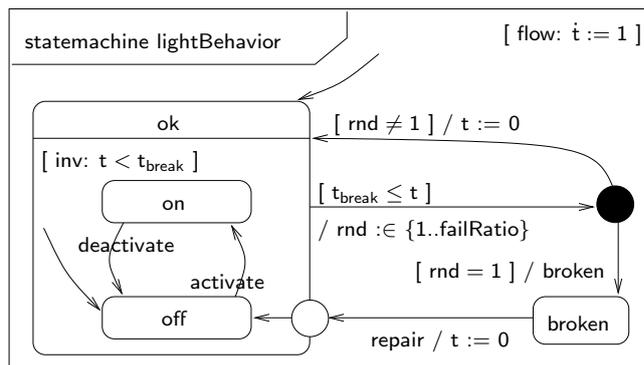


Figure C.46: Behavior of Light.

A single light can be on or off, or it can break with a certain probability. It is assumed to report the defect itself.

### C.2.20 DangerZone

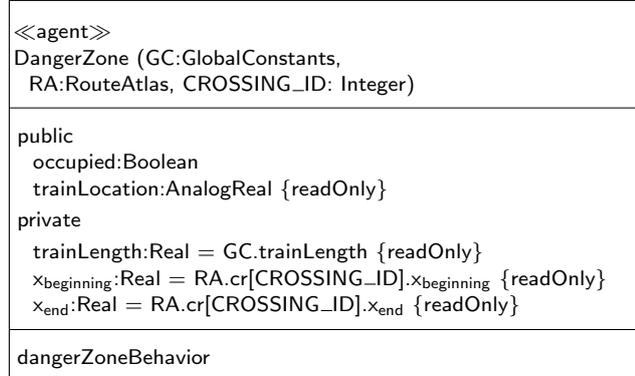


Figure C.47: Agent DangerZone.

The crossing's danger zone has a beginning position and an end position on the track, as well as the current occupancy state `occupied`. The top-level mode `dangerZoneBehavior` defines how `occupied` is determined from the length and position of the train.

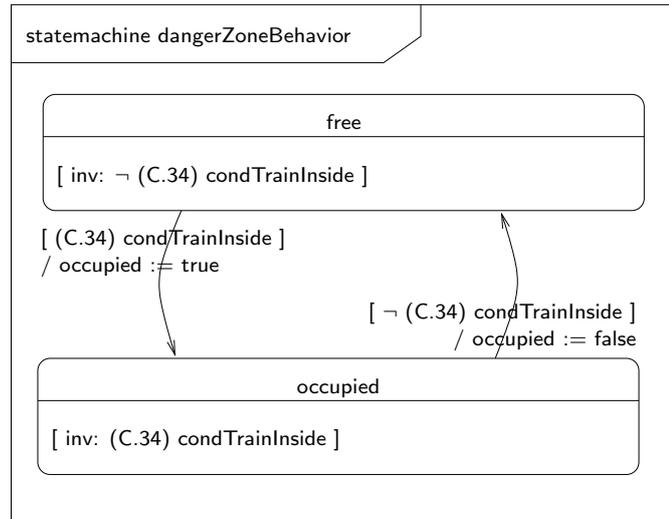


Figure C.48: Behavior of DangerZone.

$$(C.34) \text{ condTrainInside} \equiv x_{beginning} \leq trainLocation \wedge trainLocation - trainLength \leq x_{end}$$

The crossing's danger zone is either occupied (the train has entered the danger zone but not left yet) or free.

## C.2.21 LogicalGateBarrier

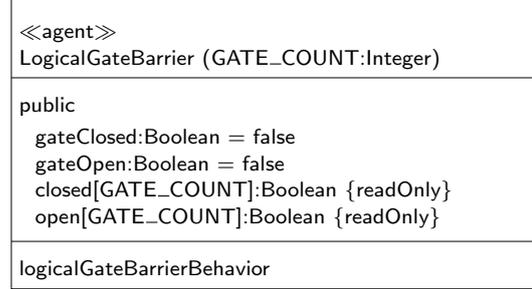


Figure C.49: Agent LogicalGateBarrier.

The logical gate barrier maps the opening status of GATE\_COUNT gates to one logical gate.

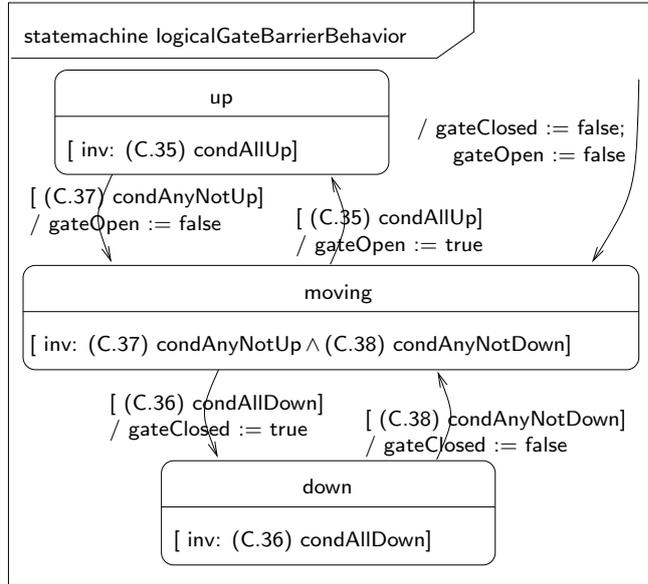


Figure C.50: Behavior of LogicalGateBarrier.

$$(C.35) \text{ condAllUp} \equiv \forall i \in \{0..GATE\_COUNT - 1\} \bullet open[i]$$

$$(C.36) \text{ condAllDown} \equiv \forall i \in \{0..GATE\_COUNT - 1\} \bullet closed[i]$$

$$(C.37) \text{ condAnyNotUp} \equiv \exists i \in \{0..GATE\_COUNT - 1\} \bullet \neg open[i]$$

$$(C.38) \text{ condAnyNotDown} \equiv \exists i \in \{0..GATE\_COUNT - 1\} \bullet \neg closed[i]$$

The logicalGateBarrierBehavior merges the status variables gateOpen and gateClosed of several gates to one instance each. Thus, the logical gate is open iff all gates are open, and it is closed, iff all gates are closed.

## C.2.22 SwitchOffSensor

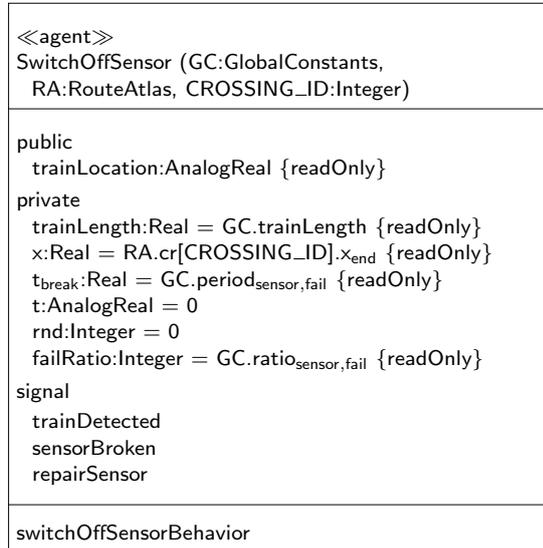


Figure C.51: Agent SwitchOffSensor.

This agent sends the `trainDetected` signal if the train passes its location `x`. The sensor reports if it becomes broken, and it can be repaired again.

The behavior is defined by `switchOffSensorBehavior`.

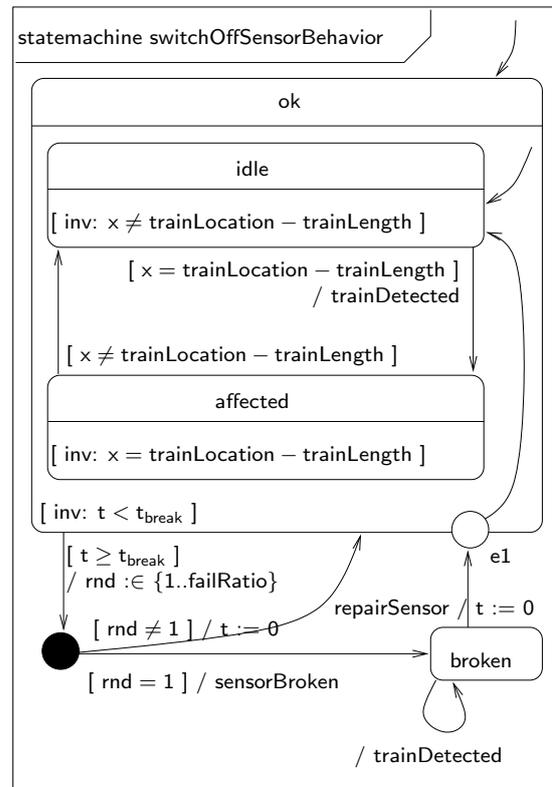


Figure C.52: Behavior of SwitchOffSensor.

Here, the expected behavior of the switch-off sensor is specified: Normally, it is idle, waiting for the train to pass the sensor’s location  $x$ . Exactly when the tail of the train is at  $x$ , the sensor generates the train detection signal. With a certain probability, the sensor can become faulty – in this case, the sensor reports its failure and afterwards generates the train detection signal at random until it is repaired again.

### C.2.23 Gate

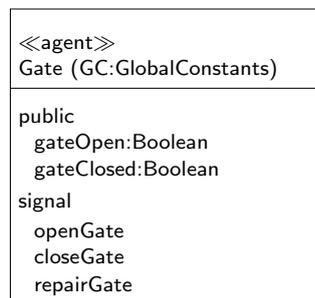


Figure C.53: Agent Gate.

The gate provides two variables that denote its current state: open, closed, or moving. Internally, the barrier has an opening angle within an interval of

possible angles. Signals that trigger the opening and closing are accepted, as well as the repair of the gate itself.

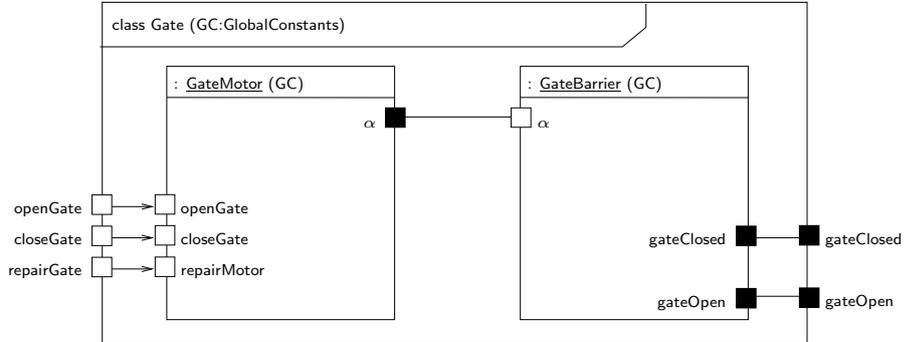


Figure C.54: Structure of Gate.

The motor and the barrier of the gate share the opening angle of the barrier, which is controlled by the motor. Open (close) gate requests are directed to the motor; the open (close) result is provided by the barrier.

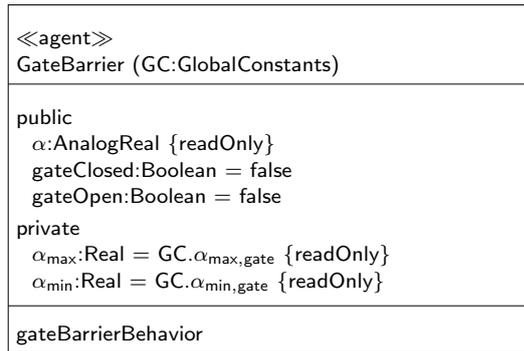


Figure C.55: Agent GateBarrier.

The gate barrier provides an opening angle within an interval of possible angles. Two signals represent the successful opening and closing of the barrier. The barrier's behavior is defined by the top-level mode gateBarrierBehavior.

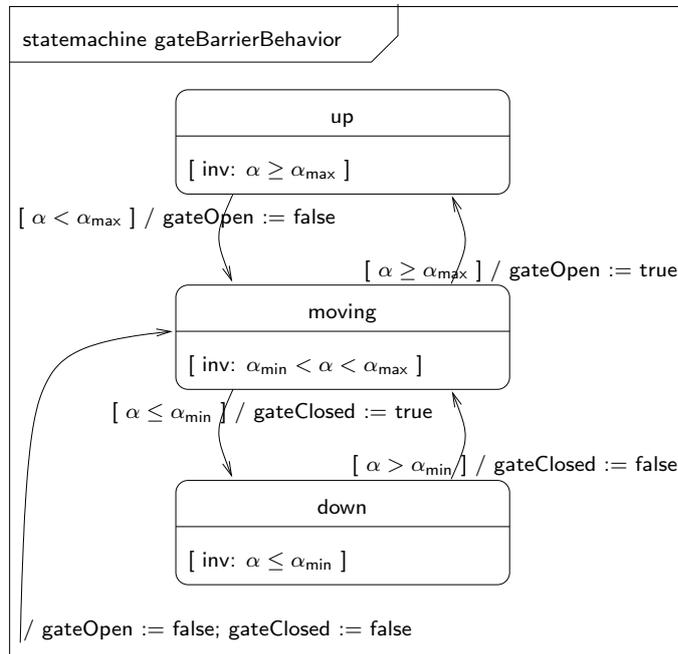


Figure C.56: Behavior of GateBarrier.

This statemachine defines the expected behavior of one gate barrier (independent of the gate motor). Subject to its opening angle  $\alpha$ , its mode is one of {up, down, moving}. When reaching up or down, a corresponding signal is generated. Here it is not specified, how  $\alpha$  changes, this is controlled by the gate motor.

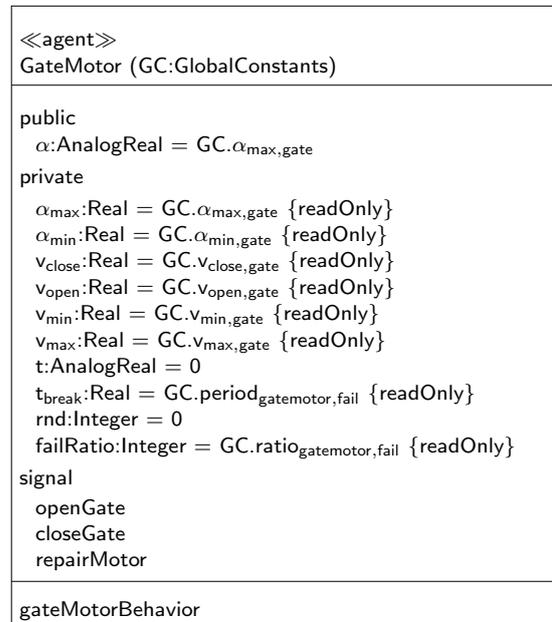


Figure C.57: Agent GateMotor.

The motor of the gate controls the actual opening angle  $\alpha$  of the gate barrier. It is bounded by the minimum and maximum opening angle. For normal behavior, opening and closing angular velocities  $v_{\text{open}}$  and  $v_{\text{close}}$  are defined. In case of failure the motor is assumed to apply a velocity within  $[v_{\text{min}}, v_{\text{max}}]$ . The motor is triggered by two signals in order to open or close the gate. Further, the motor can be repaired.

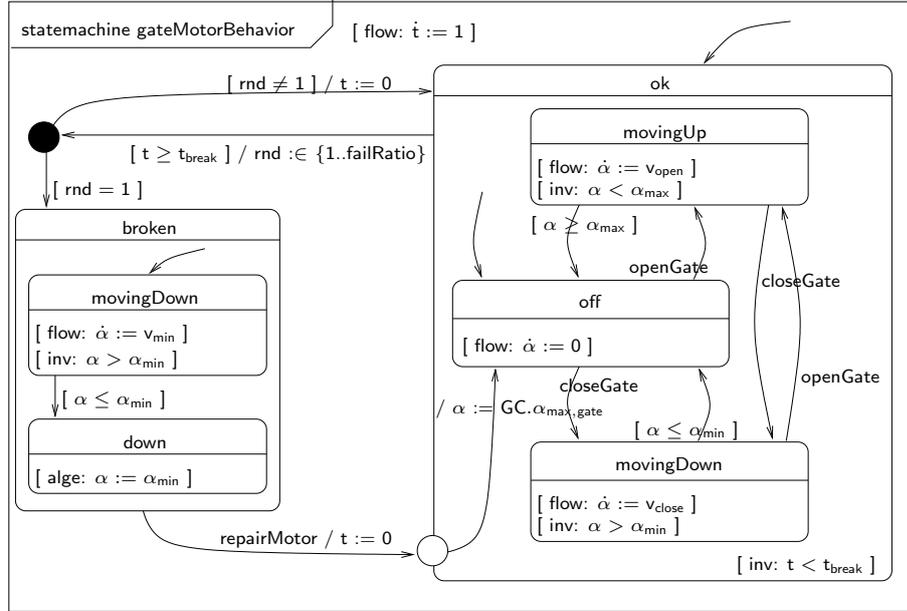


Figure C.58: Behavior of GateMotor.

The gate motor is assumed to modify the opening angle  $\alpha$  of the gate barrier by pre-defined angle velocities  $v_{\text{open}}$  and  $v_{\text{close}}$ . It is triggered by the signals `openGate` and `closeGate`. As soon as the maximum or minimum angle is reached, it switches off automatically.

The motor can break with a certain probability – in this case, it is assumed to be applying the minimum angular velocity, until the barrier is closed.

### C.2.24 CrossingController

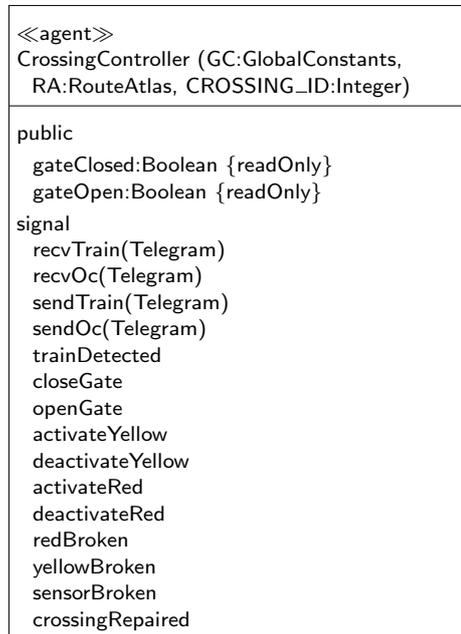


Figure C.59: Agent CrossingController.

The crossing controller gets the relevant status information from the crossing equipment: the opening status from the (logical) gate, the switch-off signal from the train sensor, and failure notifications from the lights and the sensor. It actuates the gates and the lights. Information is exchanged with the train and the operations center by means of radio telegrams.

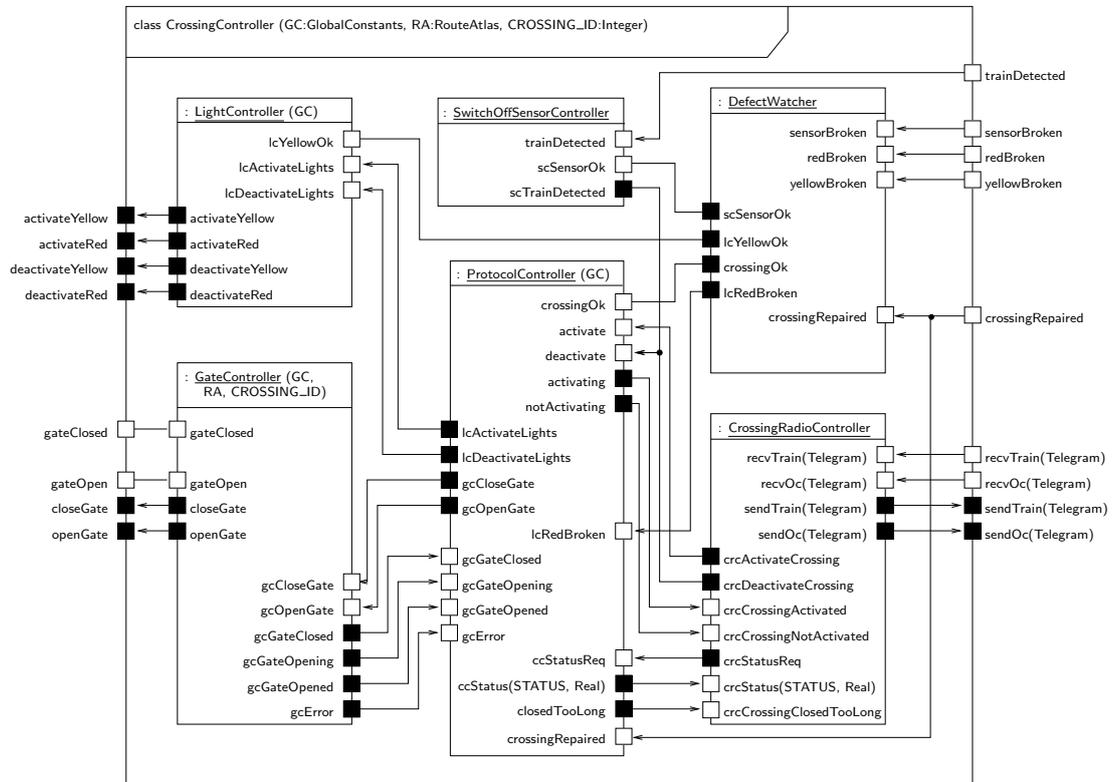


Figure C.60: Structure of CrossingController.

The crossing controller consists of several controller components:

*Switch-off sensor controller* This component controls the reception of the switch-off signal.

*Crossing radio controller* This is the radio communication part.

*Light controller* The light controller is responsible for the activation of the traffic lights.

*Gate controller* The gate controller keeps track of the gate's opening and failure status, and triggers the opening and closing of the gate.

*Protocol controller* This is the core component of the crossing controller. It coordinates the other components in order to activate and the crossing and establish the safe crossing state on train arrival, as well as the deactivation afterwards. It is split into the core protocol controller and a simple status component that provides the current crossing status.

*Defect watcher* This component receives and stores failure reports of the lights and the sensor.

### C.2.25 LightController

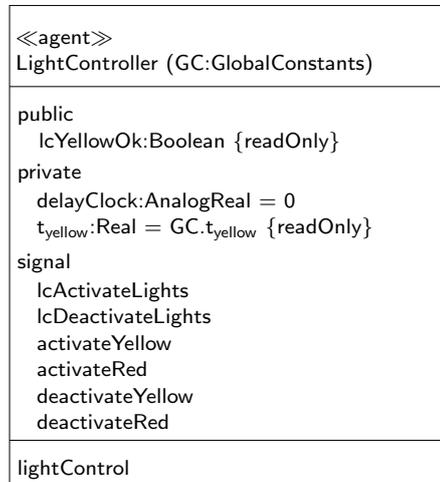


Figure C.61: Agent LightController.

The light controller controls a pair of lights (yellow and red). It activates and deactivates them individually by means of signals `activate...` and `deactivate...`. It is triggered by signals in order to start a light sequence (yellow,red) or to switch them off, respectively. The time duration for yellow (before red) and the current failure status of the yellow light is considered.

Top-level mode `lightControl` defines the behavior.

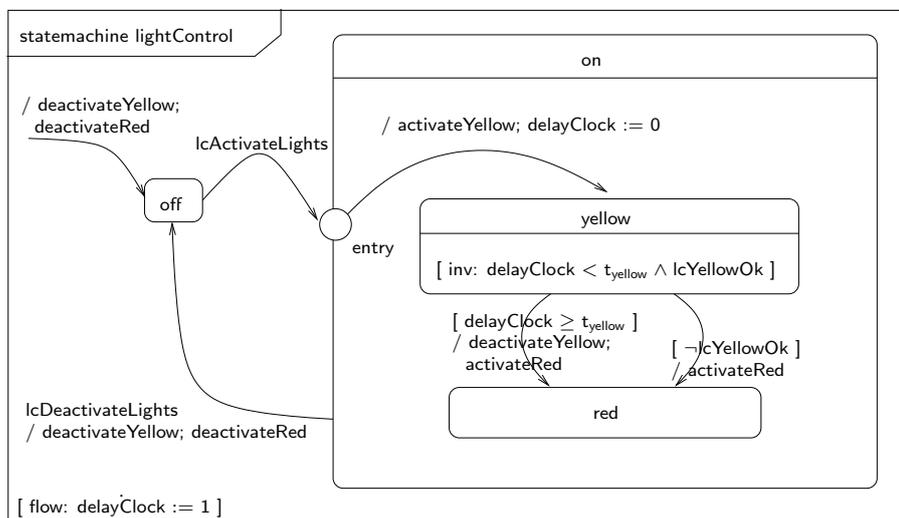


Figure C.62: Behavior of LightController.

This state machine switches on/off a pair of lights (yellow and red), triggered by the signals `lcActivateLights` and `lcDeactivateLights`. When switching on, first the yellow light is activated, and after a time duration  $t_{yellow}$  red is activated, too. If the yellow light breaks before  $t_{yellow}$  is exceeded, red is switched on immediately.

### C.2.26 GateController

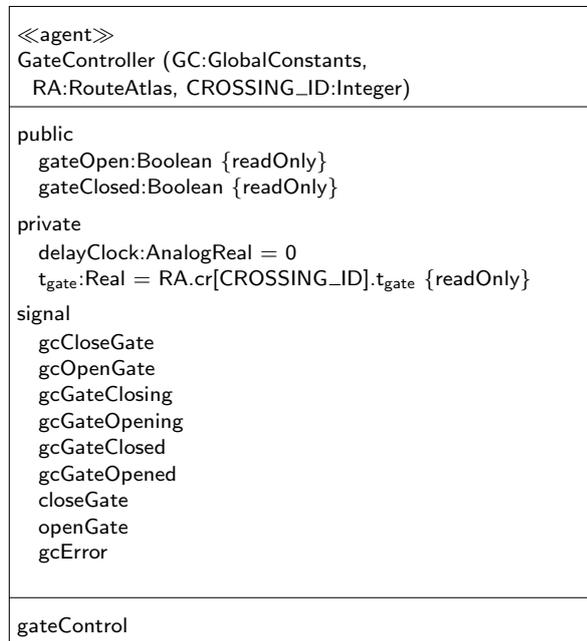


Figure C.63: Agent GateController.

This is the component that controls the gate(s). It reads the gate's actual opening status via variables `gateOpen` and `gateClosed`. The signals `openGate` and `closeGate` trigger the opening and closing of the gate. Communication with the protocol controller takes place by sending and receiving corresponding internal signals `gc...`. A maximum opening/closing time duration `tgate` is taken into account.

The behavior is defined in top-level mode `gateControl`.

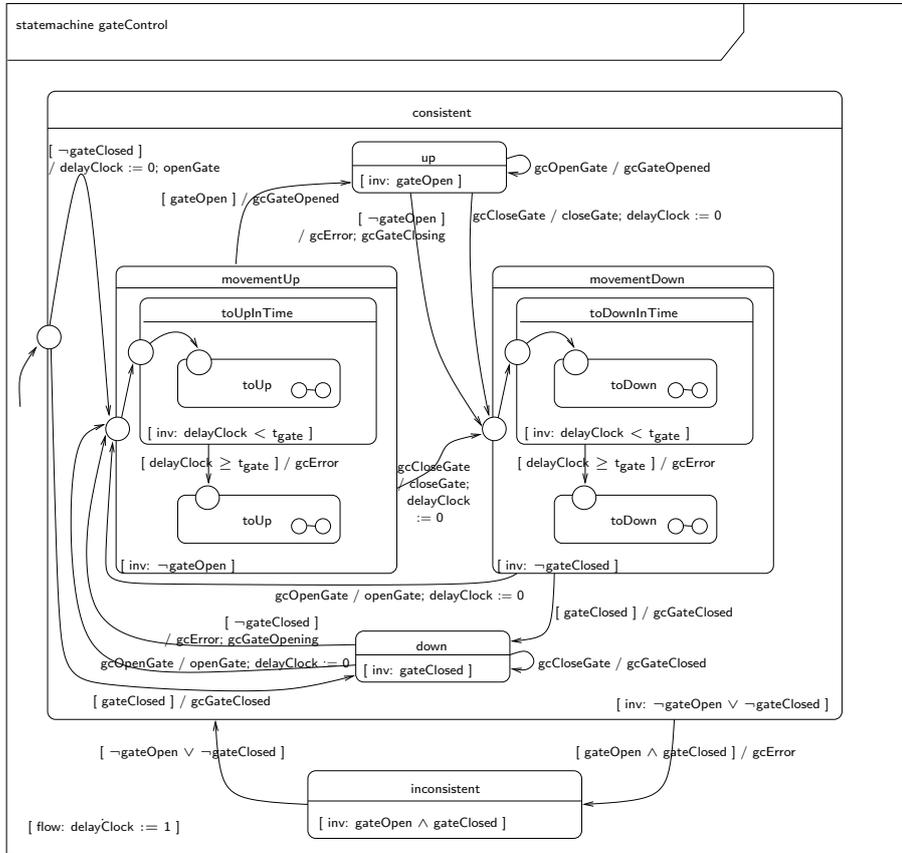


Figure C.64: Behavior of GateController.

The gate controller state machine controls a logical gate (that may consist of several actual gates). It receives gate open and close requests from the protocol controller (`gcOpenGate`, `gcCloseGate`) and triggers the corresponding actions of the gate. Further, it reports the beginning of the gate’s opening and closing activities as well as their successful termination (`gcGateOpening`, `gcGateClosing`, `gcGateClosed`, `gcGateOpened`).

The gate controller observes furthermore, if the gate is not exceeding a maximum opening/closing time duration  $t_{gate}$ . If it does, an error is reported. While the gate is moving, it is allowed to change the direction of movement, as long as  $t_{gate}$  is not exceeded. But an error is reported, if the gate reaches the open (closed) status again while closing (opening).

The controller reports successful opening (closing) immediately if the gate is open (closed) yet. If the gate is opening (closing), a close (open) request leads to abortion of the opening (closing) activity; the closing (opening) activity is started. In this case, the maximum time duration is also  $t_{gate}$ .

The gate controller is provided the boolean variables `gateOpen` and `gateClosed` by the gate which should not be true at the same time. If they are, the gate must be defect.

There are two different ways of initializing the gate controller: if the gate is closed, the controller’s initial mode is set correspondingly. Otherwise, the gate

will be opened.

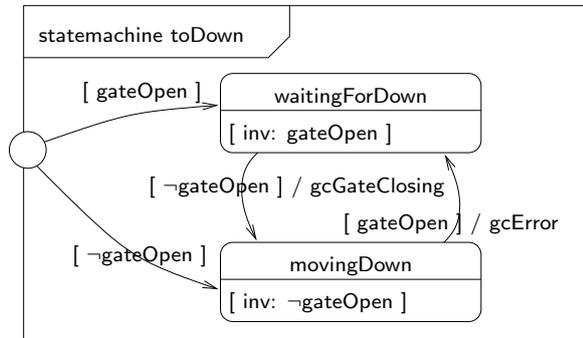


Figure C.65: Submode toDown.

See statemachine gateControl.

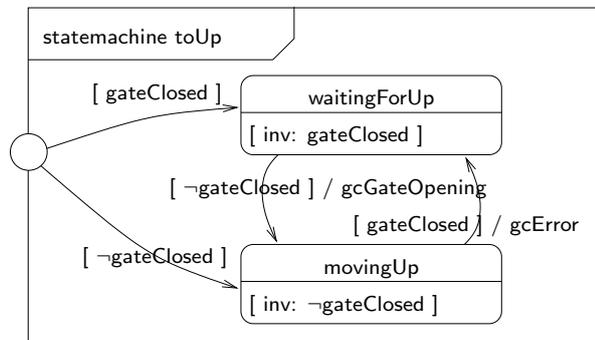


Figure C.66: Behavior of Submode toUp..

See statemachine gateControl.

### C.2.27 SwitchOffSensorController

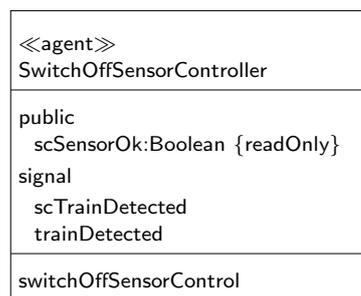


Figure C.67: Agent SwitchOffSensorController.

This agent controls the switch-off sensor. It only generates a signal `scTrainDetected` if the sensor reports the train detection (`trainDetected`) – see top-level mode `switchOffSensorControl`.

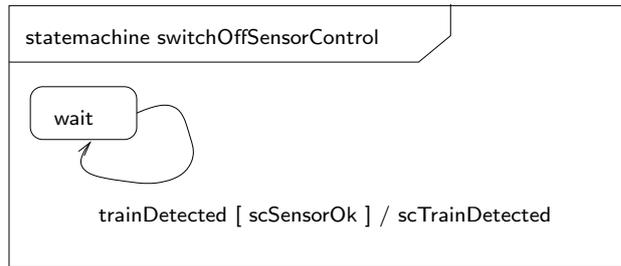


Figure C.68: Behavior of SwitchOffSensorController.

This state machine generates the signal `scTrainDetected`, whenever it receives the detection signal from the switch-off sensor, as long as the sensor is operative.

### C.2.28 Defect Watcher

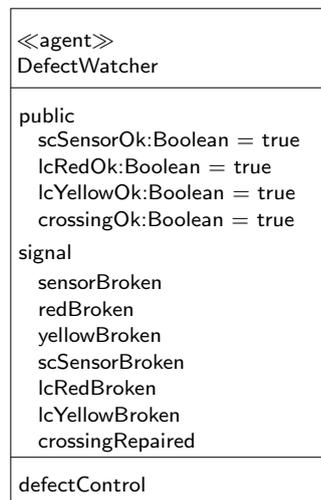


Figure C.69: Agent DefectWatcher.

The defect watcher component keeps track of the defects that are reported from the train sensor and the lights: `sensorBroken`, `redBroken`, `yellowBroken`. It stores the status (ok/defect) and generates failure signals.

Failure status is reset on the repairing of the crossing.

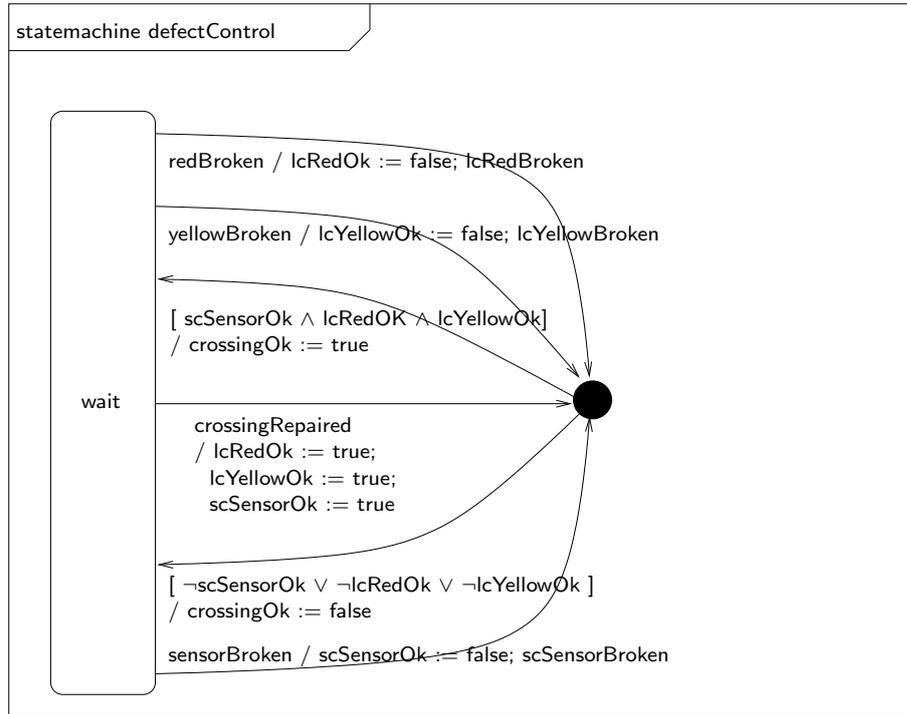


Figure C.70: Behavior of DefectWatcher.

The defect watcher coordinates the failure status of those components that report failures themselves: the lights and the train sensor. It receives their failure reports `redBroken`, `yellowBroken` and `sensorBroken`, and generates corresponding internal signals. Further, boolean variables are updated that store the actual status. `crossingOk` is defined as a shortcut for convenience.

The failure status of the gate is not observed here, this must be detected by the gate controller.

### C.2.29 CrossingRadioController

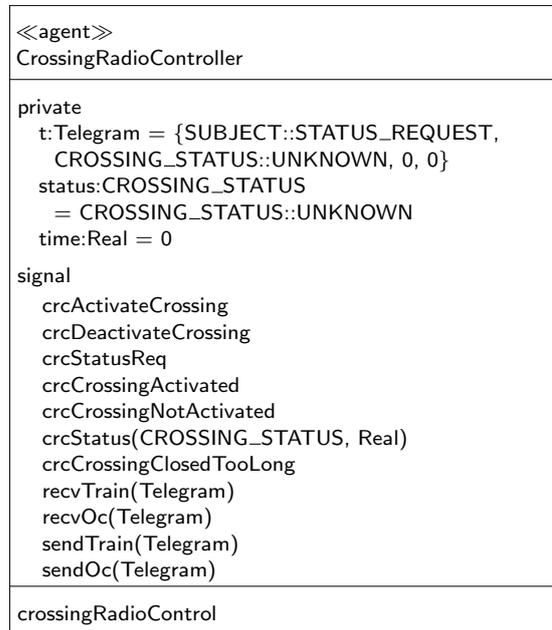


Figure C.71: Agent CrossingRadioController.

The crossing radio controller maps the dispatch and the reception of radio telegrams (rcv..., send...) to respective controller-internal signals (crc...). It is a basic agent with top-level mode crossingRadioControl.

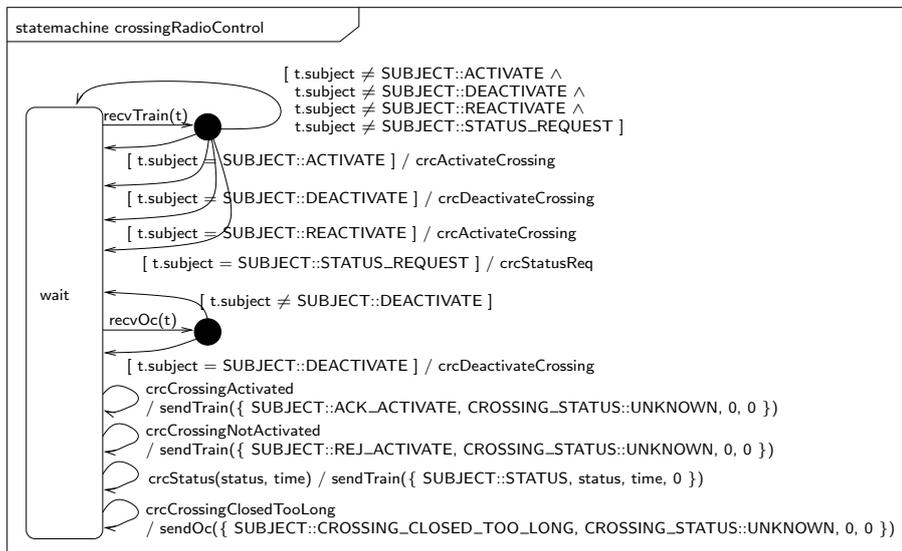


Figure C.72: Behavior of CrossingRadioController.

crossingRadioControl simply maps radio telegrams to signals and vice versa. Incoming radio telegrams are represented as signals rcvTrain(Telegram) and

recvOc(Telegram), outgoing radio telegrams are signals sendTrain(Telegram) and sendOc(Telegram). There is exactly one (full-duplex) radio channel to each the train and the operations center.

Here, no assumptions are made about how the radio channels behave, if the telegrams are actually received and how long this will take. Therefore, the send... signals represent the *departure* of the respective telegrams. The signals recv... denote the actual reception of telegrams.

### C.2.30 ProtocolController



Figure C.73: Agent ProtocolController.

This controller subsumes the core protocol controller and the protocol status controller. See ProtocolControllerCore and ProtocolStatusController for details.

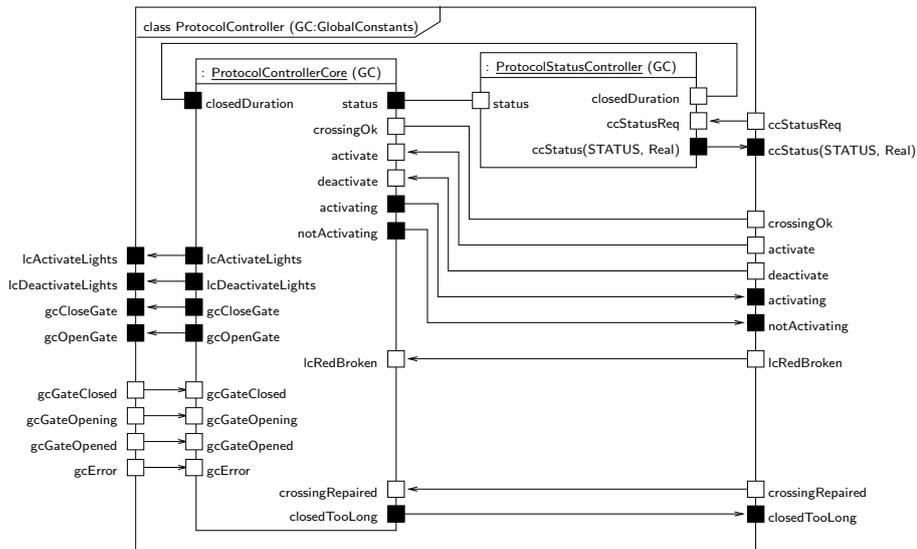


Figure C.74: Structure of ProtocolController.

The two components core controller and status controller share the crossing status and the actual closing duration, both provided by the core controller. The status controller only provides this information on request, the controlling of the crossing is done by the core controller.

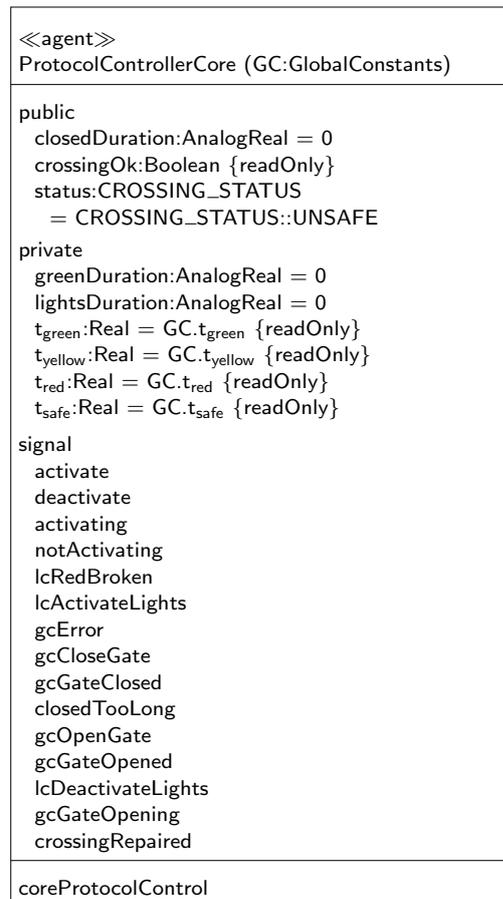


Figure C.75: Agent ProtocolControllerCore.

The protocol controller triggers the components light controller and gate controller via signals `lcActivateLights`, `lcDeactivateLights`, `gcOpenGate` and `gcCloseGate`, and gets feedback by means of `gcGateOpening`, `gcGateOpened` and `gcGateClosed`.

The failure statuses and signals of the light controller, gate controller and switch-off sensor controller have impact on the protocol. Further, several time durations are considered:

$t_{\text{green}}$  The minimum time duration for road users to cross the tracks between two subsequent crossing closings.

$t_{\text{yellow}}$  The duration for which the yellow light is active on closing the crossing.

$t_{\text{red}}$  The red light duration before the gates are closed.

$t_{\text{safe}}$  Road users are expected to cross the tracks although they are not allowed to if they have to wait too long. It is assumed, that the crossing is safe while being closed for the time duration  $t_{\text{safe}}$  – before people are becoming impatient.

The core controller is triggered by signals `activate` and `deactivate`, and responds with `activating` or `notActivating`, if any component is defect.

The actual status as well as the actual closing duration are provided to the status controller.

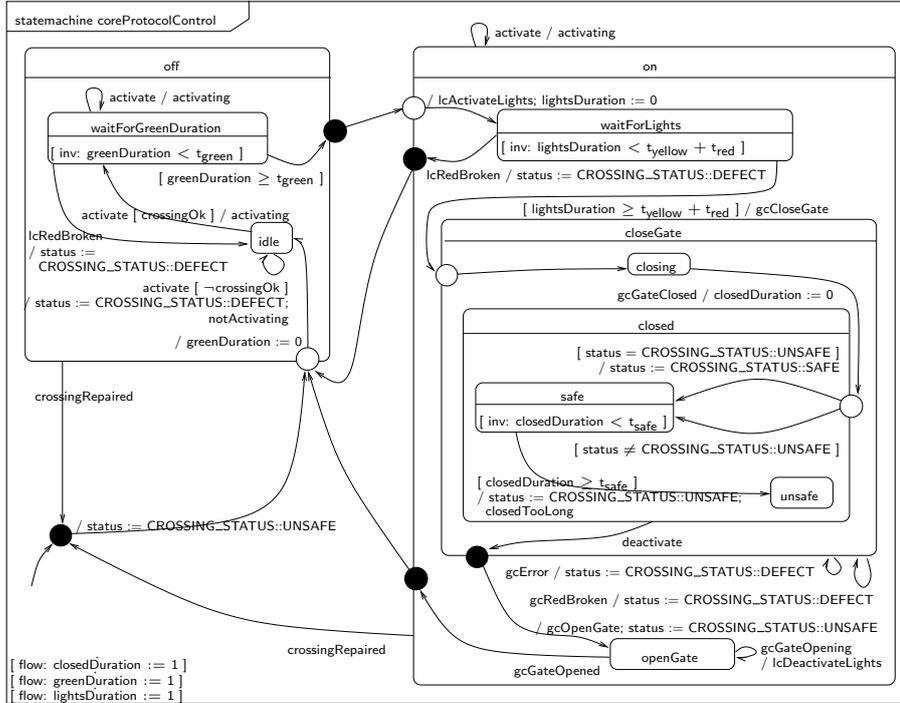


Figure C.76: Behavior of ProtocolControllerCore.

This state machine defines the core behavior of the crossing controller. It coordinates the controller components that are responsible for the lights, the gate and the switch-off sensor.

Initially, the controller is in an idle mode, waiting for the activation trigger. After reception of the activation signal and after a minimum time duration  $t_{\text{green}}$  (in order to guarantee a time interval to pass the crossing for cars, pedestrians etc.), the crossing is “activated”: first, the lights are triggered, then the closing of the gate is initiated. As soon as the gate is closed, the crossing is considered as being safe for a maximum time duration  $t_{\text{safe}}$ . Afterwards, the crossing stays closed, but is not considered as being safe anymore, because people waiting at the crossing are assumed to be impatient and to pass the crossing although they are not allowed to. Finally, the deactivation trigger of the switch-off sensor causes the gate to be opened again and the lights to be switched off. The core controller re-enters its idle mode, when the gate is open again.

If errors occur during the crossing activation protocol, the activation will be interrupted, leaving the crossing open. There is one exception: the defect of the yellow light is not critical, so activation will continue. As soon as the gate starts closing, the crossing will be closed regardless of any error, but it will not be considered as safe.

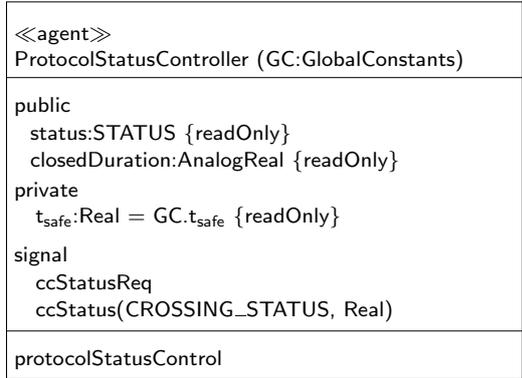


Figure C.77: Agent ProtocolStatusController.

The protocol status controller always sends the actual crossing status `status` by means of signal `ccStatus(...)` on request `ccStatusReq`. The signal includes the remaining maximum time duration for which the crossing will be safe, calculated from `tsafe` and `closedDuration`.

This is defined in `protocolStatusControl`.

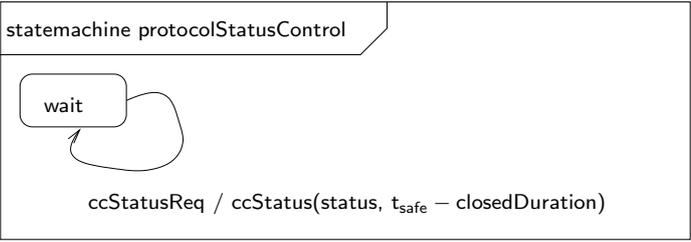


Figure C.78: Behavior of ProtocolStatusController.

The simple purpose of `smCrossingProtocolStatusController` is to create a requested status signal that contains the crossing’s status as well as the remaining time duration for which the crossing is assumed to be safe.

**C.2.31 OperationsCenter**

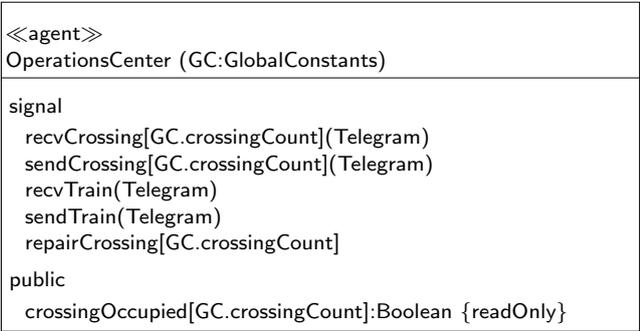


Figure C.79: Agent OperationsCenter.

The operations center communicates with the train and the crossings via the sending and reception of radio telegrams. If required, it repairs specific crossings or inquires their occupancy status.

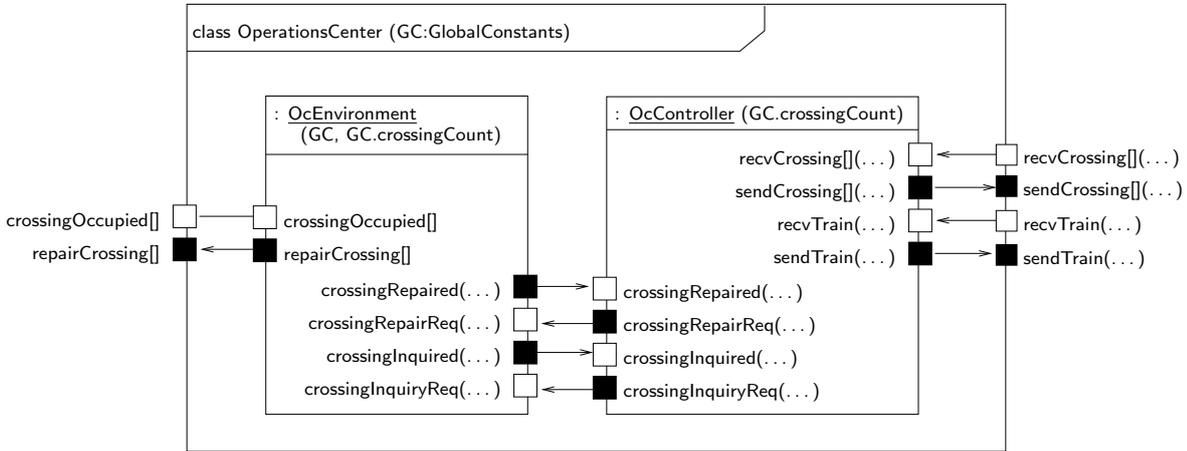


Figure C.80: Structure of OperationsCenter.

The operations center consists of a controller that automatically manages status messages and requests from and to the train and the crossings, and an abstraction of the operations center environment including operations center staff. The operations center environment’s responsibility includes the manual checking whether crossings are occupied as well as repairing defective equipment.

### C.2.32 OcEnvironment

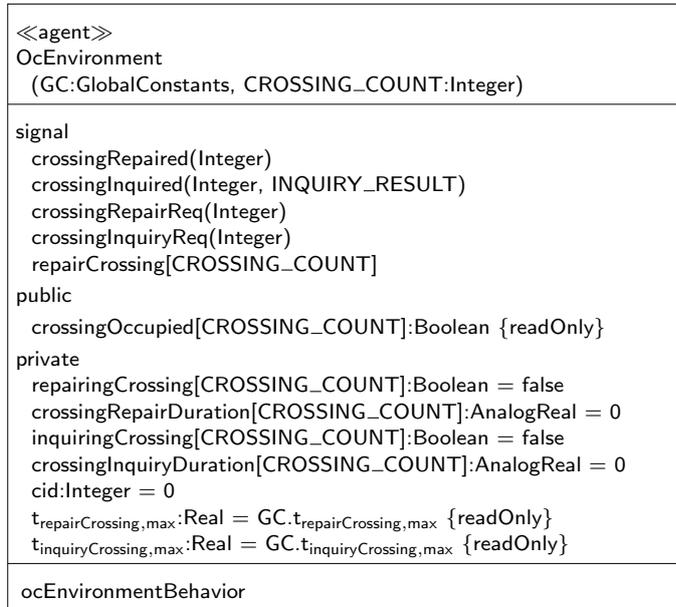


Figure C.81: Agent OcEnvironment.

The operations center environment will repair (send a `repairCrossing` signal) or inquire a specific crossing (check `crossingOccupied`) on request (`crossingRepairReq(...)`, `crossingInquiryReq(...)`).

The repair as well as the inquiry take some time, bounded by  $t_{\dots, \max}$ .

The environment's behavior is defined in top-level mode `ocEnvironment`.

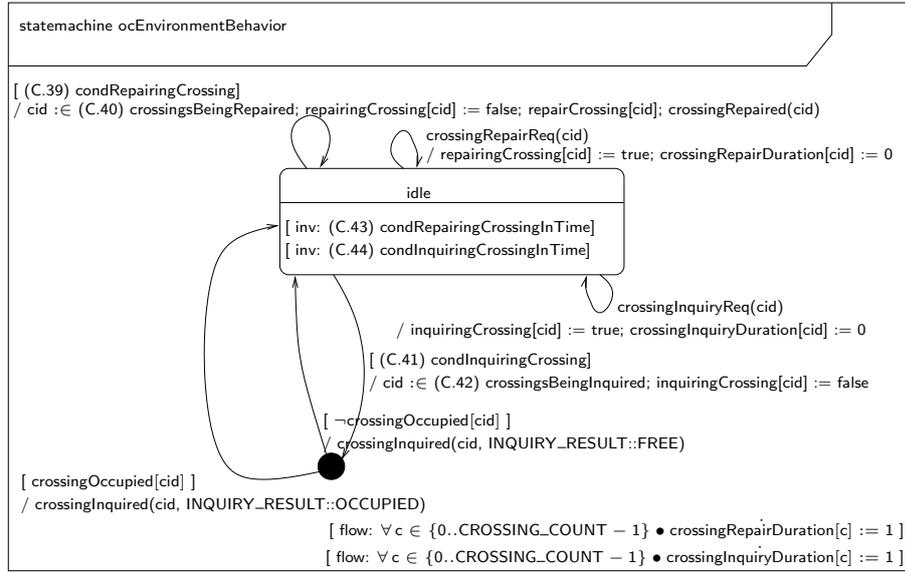


Figure C.82: Behavior of `OcEnvironment`.

$$(C.39) \text{condRepairingCrossing} \equiv \exists c \in \{0..CROSSING\_COUNT - 1\} \bullet \text{repairingCrossing}[c]$$

$$(C.40) \text{crossingsBeingRepaired} \equiv \{c \in \{0..CROSSING\_COUNT - 1\} \mid \text{repairingCrossing}[c]\}$$

$$(C.41) \text{condInquiringCrossing} \equiv \exists c \in \{0..CROSSING\_COUNT - 1\} \bullet \text{inquiringCrossing}[c]$$

$$(C.42) \text{crossingsBeingInquired} \equiv \{c \in \{0..CROSSING\_COUNT - 1\} \mid \text{inquiringCrossing}[c]\}$$

$$(C.43) \text{condRepairingCrossingInTime} \equiv \forall c \in \{0..CROSSING\_COUNT - 1\} \bullet (\neg \text{repairingCrossing}[c] \vee \text{crossingRepairDuration}[c] \leq t_{\text{repairCrossing}, \max})$$

$$(C.44) \text{condInquiringCrossingInTime} \equiv \forall c \in \{0..CROSSING\_COUNT - 1\} \bullet (\neg \text{inquiringCrossing}[c] \vee \text{crossingInquiryDuration}[c] \leq t_{\text{inquiryCrossing}, \max})$$

The operations center's environment models the manual repairing and inquiring activities. As soon as a request is received the respective activity is started. Maximum time durations for both kinds of activities are assumed ( $t_{\text{repairCrossing}, \max}$ ,  $t_{\text{inquiryCrossing}, \max}$ ). In case of inquiry, the crossing's occupancy status is returned.

### C.2.33 OcController

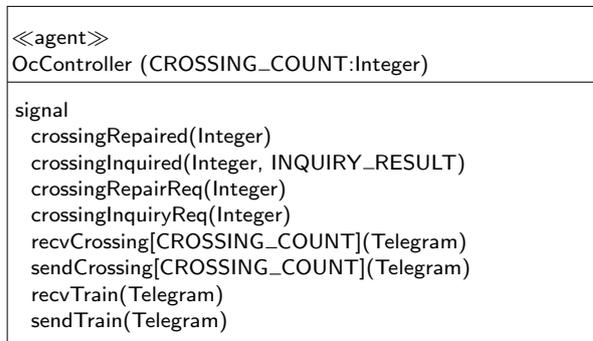


Figure C.83: Agent OcController.

The operations center controller communicates with the train and the crossings via the sending and reception of radio telegrams (`send...`, `rcv...`). To its environment, it requests repair and inquiry actions concerning specific crossings (`crossing...Req`). When finished, the environment states `crossingRepaired(...)` or `crossingInquired(...)`, respectively.

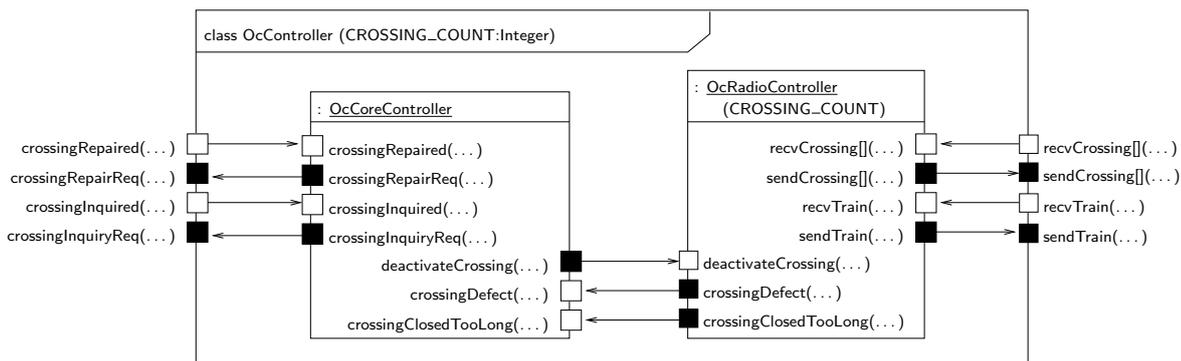


Figure C.84: Structure of OcController.

The operations center controller is divided into its core controller and its radio controller. The core controller manages the interdependencies between status messages and requests concerning the train and the crossings, as well as required actions of the operations center environment and their results. The radio controller defines the radio communication interface of the operations center controller.

## C.2.34 OcCoreController

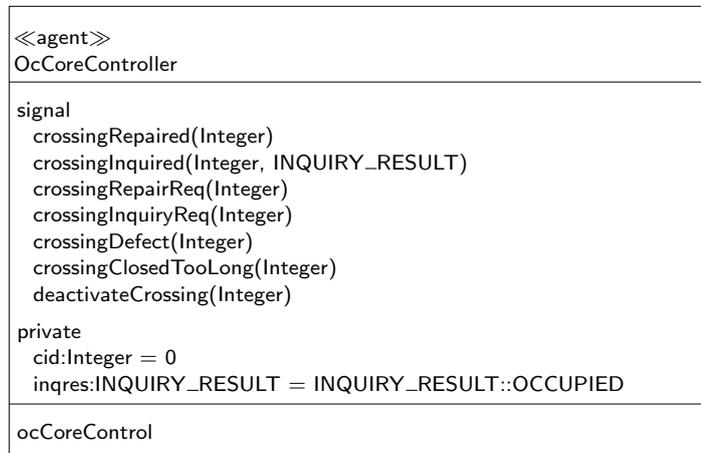


Figure C.85: Agent OcCoreController.

The operations center core controller receives signals that denote crossing defects or the violation of the maximum safe closing time from the radio controller. To its environment, it requests a repair or an inquiry action for the affected crossing (`crossing...Req`). When finished, the environment states `crossingRepaired(...)` or `crossingInquired(...)`, respectively.

A repair request simply triggers the manual repairing of the crossing, an inquiry request is sent if a crossing is closed too long, in order to assure that the crossing is currently not occupied by the train and can be opened again. This is triggered via signal `deactivateCrossing(...)`.

`ocCoreControl` defines the behavior of the core controller.

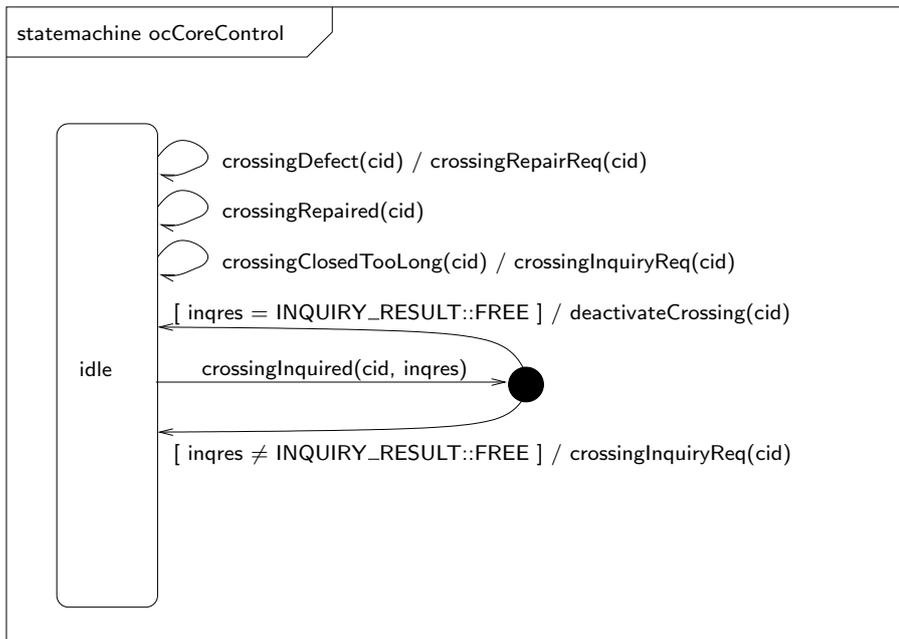


Figure C.86: Behavior of OcCoreController.

The operations center's core controller only provides basic functionality concerning the recovery of normal operation:

- If any defect is reported, repairing is initiated. No action is required when repairing finishes.
- If any crossing is closed too long, a crossing inquiry is triggered. Subject to the crossing's danger zone status, the inquiry receipt leads to the deactivation of the respective crossing or to a repetition of the inquiry.

### C.2.35 OcRadioController

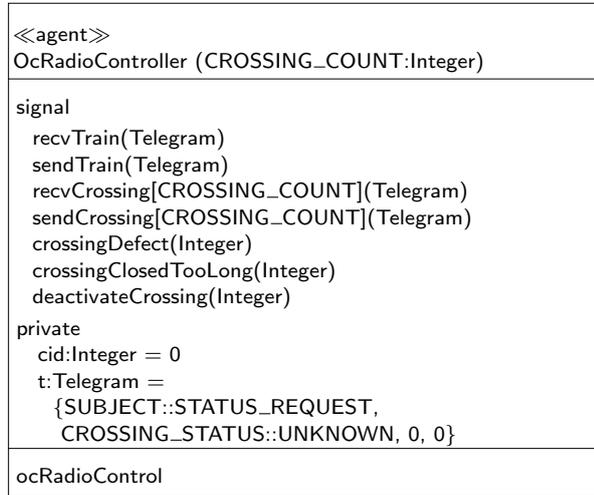


Figure C.87: Agent OcRadioController.

The operations center radio controller maps the controller-internal signal `deactivateCrossing(...)` to the sending of the respective radio telegram, and maps the reception of specific radio telegrams to the controller-internal signals `crossingDefect(...)` and `crossingClosedTooLong(...)`.

`ocRadioControl` is the behavior specification.

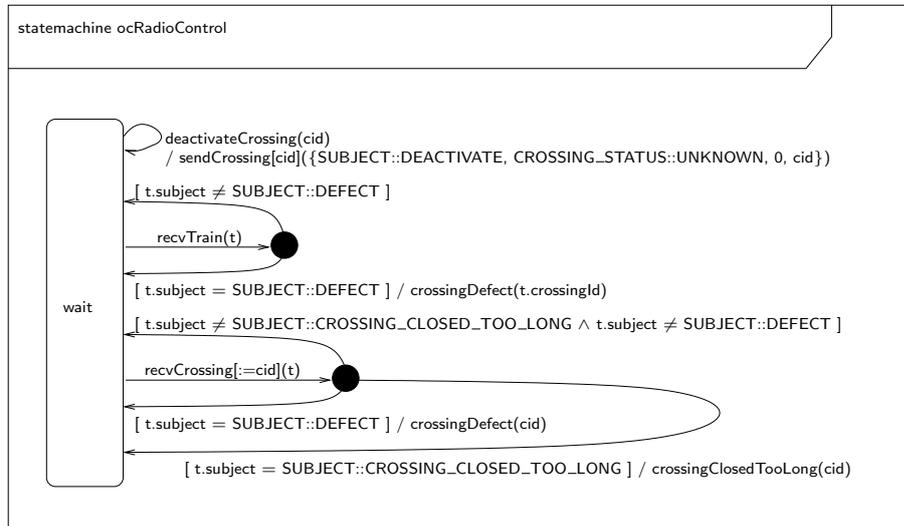


Figure C.88: Behavior of OcRadioController.

This statemachine maps status messages and requests to the respective radio messages. See also section C.2.29: `crossingRadioControl`.

### C.2.36 RadioChannelTrainCrossing

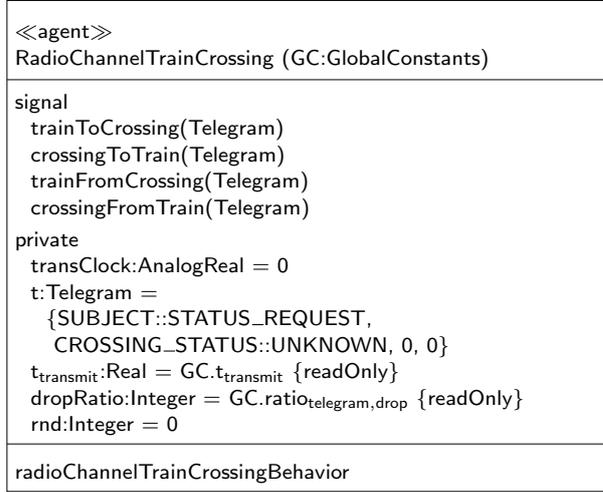


Figure C.89: Agent RadioChannelTrainCrossing.

The radio channel between the train and the crossing contains signals that denote the sending of a radio message from the train to the crossing, and the sending from the crossing to the train, as well as the respective reception of a radio message.

A maximum transmission time  $t_{transmit}$  for radio messages is assumed.

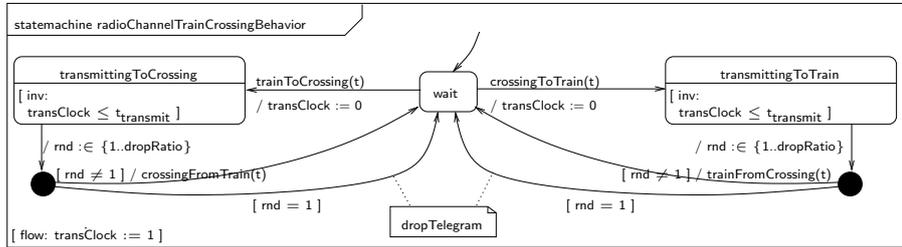


Figure C.90: Behavior of RadioChannelTrainCrossing.

The radio channel that links the train and the crossing is assumed to be unreliable – messages that are sent at either side may be silently lost. The time duration needed for a successful transmission has an upper bound  $t_{transmit}$ .

### C.2.37 RadioChannelTrainOc

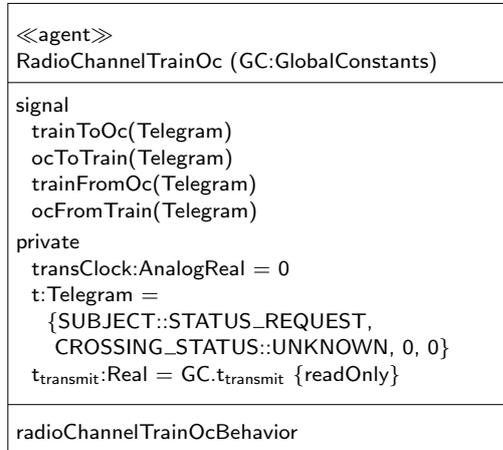


Figure C.91: Agent RadioChannelTrainOc.

The radio channel between the train and the operations center contains signals that denote the sending of a radio message from the train to the operations center, and the sending from the operations center to the train, as well as the respective reception of a radio message.

A maximum transmission time  $t_{\text{transmit}}$  for radio messages is assumed.

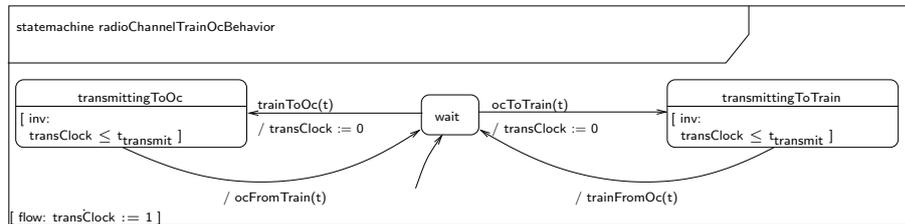


Figure C.92: Behavior of RadioChannelTrainOc.

The radio channel that links the train and the operations center is assumed to be reliable – every message that is sent at either side is received on the other. The time duration needed for transmission has an upper bound  $t_{\text{transmit}}$ .

### C.2.38 RadioChannelCrossingOc

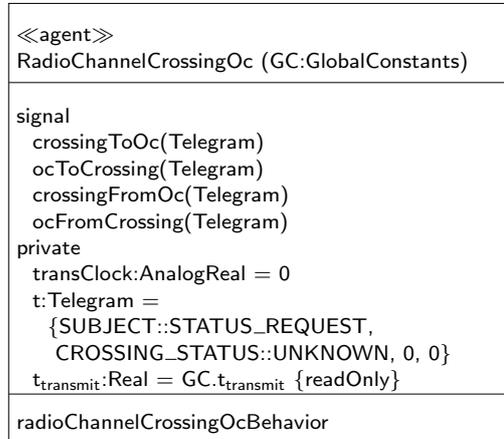


Figure C.93: Agent RadioChannelCrossingOc.

The radio channel between one crossing and the operations center contains signals that denote the sending of a radio message from the crossing to the operations center, and the sending from the operations center to the crossing, as well as the respective reception of a radio message.

A maximum transmission time  $t_{\text{transmit}}$  for radio messages is assumed.

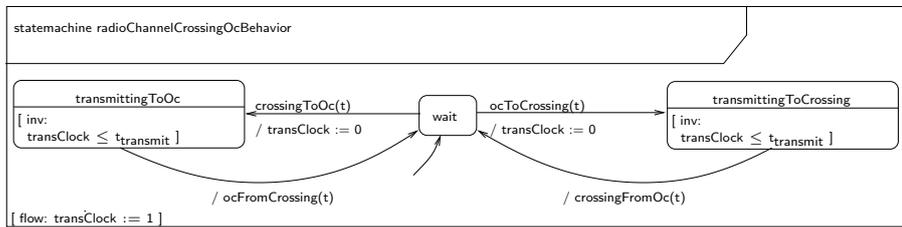


Figure C.94: Behavior of RadioChannelCrossingOc.

The radio channel that links the crossing and the operations center is assumed to be reliable – every message that is sent at either side is received on the other. The time duration needed for transmission has an upper bound  $t_{\text{transmit}}$ .



# Appendix D

## Code Examples

In this chapter, excerpts from the generated code of the case study Radio-Based Train Control are presented. The given code is created by the implementation of the transformation  $\Phi_{HUMML}$  and is a C++ variant of the formally defined HL<sup>3</sup> model.

This chapter contains excerpts of a C++ program which corresponds to an HL<sup>3</sup> program – the transformation result of the HybridUML model of the case study “Radio-Based Train Control”. Its purpose is to give an impression of the generated code which is produced by the implementation of the transformation  $\Phi_{HUMML}$ . From the full transformation, two aspects are presented here:

*HL<sup>3</sup> Model Definition* Corresponding to the definition of the HL<sup>3</sup> model  $c_{spec} \in CONST$  as discussed in section 5.4, *entities* are defined here as C++ objects. They are instantiated, and some of their *dependencies* are directly expressed by object references.

*Code Create for Expression Nodes* According to section 5.5, from expressions within their specific context, programs  $p \in Program$  result that actually calculate the expressions at run-time. The implemented transformation creates respective C++ operation definitions.

As a subset of the HybridUML model’s functionality, the specification of the basic agent BrakePointController is chosen, which is discussed in section 1.3.2. The complete model definition contains further details, given in section C.2.17. The agent definitions of Fig. 1.11 and Fig. C.39, as well as the behavior definitions in Fig. 1.13 and Fig. C.40 are identical.

We repeat the behavior definition `brakePointControl` here (Fig. D.1) for the comprehensibility of the presented C++ code below. For the same reason, the comments within the code have been manually improved.

Technically, we have created an instance of the implemented HybridUML Mathematical Meta-Model that corresponds to the graphical model of appendix C.2. This has been used as input for the implemented transformation.

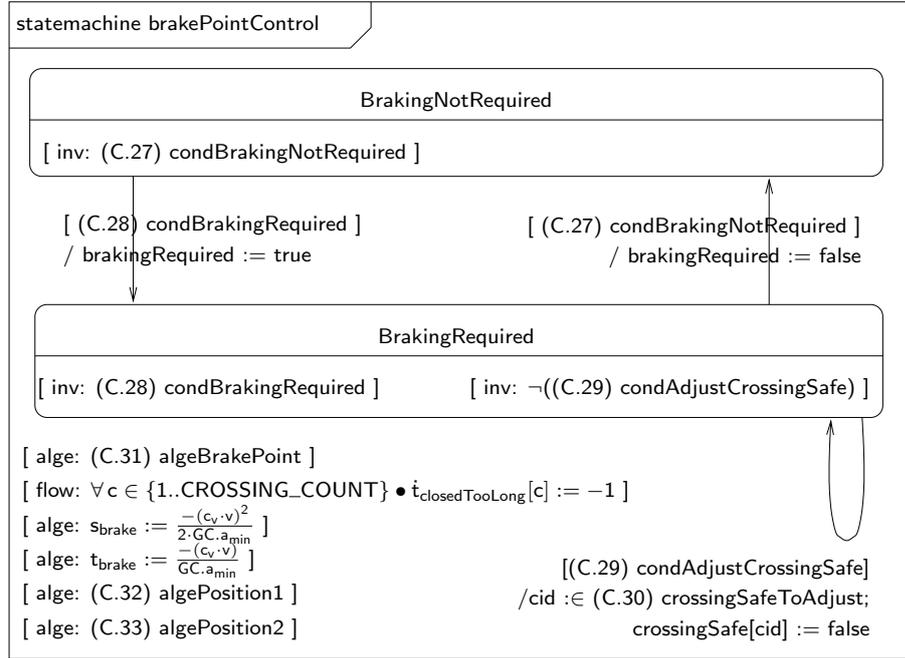


Figure D.1: Behavior of agent BrakePointController.

## D.1 HL<sup>3</sup> Model

The C++ code that represents the particular HL<sup>3</sup> model  $c_{tc} \in CONST$  for the Radio-Based Train Control is given by the conventional top-level operation `main`. This is divided into three parts: (1) the definition and initialization of channels and ports, (2) the definition of abstract machines and their internal static structure, along with the instantiation of flows and transitions that are associated with the respective abstract machine, and (3) the instantiation of the selector, as well as the scheduler.

### D.1.1 Definition of Channels and Ports

Each channel and port, defined by  $chan(c_{tc})$  or  $port(c_{tc})$ , is represented as a C++ object. They are created, and the mapping from ports to channels  $chan_{port}(c_{tc})$  is defined by object references. Afterwards, the initial values  $initval_{port}(c_{tc})$  for ports are inserted into the corresponding channels.

In the code below, channels and ports for the variables `brakingRequired` and `brakePoint` of the agent `BrakePointController` are shown. Corresponding to the multiplicities, there are a single channel for `brakingRequired`, but 11 ones for `brakePoint`:

```
int main (int argc, char **argv) {
    /***/
    /* create channels */
    /***/
    Channel::brakingRequired_36 = new hl3::TypedChannel<Boolean>(2);
```

```

//
Channel::brakePoint_53 = new hl3::TypedChannel<AnalogReal>(1);
Channel::brakePoint_54 = new hl3::TypedChannel<AnalogReal>(1);
Channel::brakePoint_55 = new hl3::TypedChannel<AnalogReal>(1);
Channel::brakePoint_56 = new hl3::TypedChannel<AnalogReal>(1);
Channel::brakePoint_57 = new hl3::TypedChannel<AnalogReal>(1);
Channel::brakePoint_58 = new hl3::TypedChannel<AnalogReal>(1);
Channel::brakePoint_59 = new hl3::TypedChannel<AnalogReal>(1);
Channel::brakePoint_60 = new hl3::TypedChannel<AnalogReal>(1);
Channel::brakePoint_61 = new hl3::TypedChannel<AnalogReal>(1);
Channel::brakePoint_62 = new hl3::TypedChannel<AnalogReal>(1);
Channel::brakePoint_63 = new hl3::TypedChannel<AnalogReal>(1);
/* ... */

/*****
/* create ports */
*****/
Port::brakingRequired_360x853fd68 = new hl3::TypedPort<Boolean>();
//
Port::brakePoint_530x853ff80 = new hl3::TypedPort<AnalogReal>();
Port::brakePoint_540x853ff90 = new hl3::TypedPort<AnalogReal>();
Port::brakePoint_550x853ffa0 = new hl3::TypedPort<AnalogReal>();
Port::brakePoint_560x853ffb0 = new hl3::TypedPort<AnalogReal>();
Port::brakePoint_570x853ffc0 = new hl3::TypedPort<AnalogReal>();
Port::brakePoint_580x853ffd0 = new hl3::TypedPort<AnalogReal>();
Port::brakePoint_590x853ffe0 = new hl3::TypedPort<AnalogReal>();
Port::brakePoint_600x853fff0 = new hl3::TypedPort<AnalogReal>();
Port::brakePoint_610x8540000 = new hl3::TypedPort<AnalogReal>();
Port::brakePoint_620x8540010 = new hl3::TypedPort<AnalogReal>();
Port::brakePoint_630x8540020 = new hl3::TypedPort<AnalogReal>();
/* ... */

/*****
/* initialize ports */
*****/
brakingRequired_360x853fd68->setChannel(*Channel::brakingRequired_36);
//
brakePoint_530x853ff80->setChannel(*Channel::brakePoint_53);
brakePoint_540x853ff90->setChannel(*Channel::brakePoint_54);
brakePoint_550x853ffa0->setChannel(*Channel::brakePoint_55);
brakePoint_560x853ffb0->setChannel(*Channel::brakePoint_56);
brakePoint_570x853ffc0->setChannel(*Channel::brakePoint_57);
brakePoint_580x853ffd0->setChannel(*Channel::brakePoint_58);
brakePoint_590x853ffe0->setChannel(*Channel::brakePoint_59);
brakePoint_600x853fff0->setChannel(*Channel::brakePoint_60);
brakePoint_610x8540000->setChannel(*Channel::brakePoint_61);
brakePoint_620x8540010->setChannel(*Channel::brakePoint_62);
brakePoint_630x8540020->setChannel(*Channel::brakePoint_63);
/* ... */

/*****
/* initialize channels */
*****/
// Unrestricted visibility
// => all ports get the same initial values (per channel)
hl3::VisibilityAttribute globallyVisibleNow(hl3::TimeService::getHL3Time(),
&hl3::Port::getAllPortIds());

hl3::VisibilitySet &visSet = hl3::VisibilitySet::create(1);
visSet.append(&globallyVisibleNow);
f // initialize channel "brakingRequired_36"
  hl3::byteSeq_t &byteSeq =
    hl3::byteSeq_t::create(sizeof(Boolean), sizeof(Boolean));

```

```

Boolean &value = *(Boolean*)byteSeq.getValue();
/* begin: initialization of value */
value = false;
/* end: initialization of value */
Channel::brakingRequired_36->putInit(byteSeq, visSet);
hl3::byteSeq_t::destroy(byteSeq);
}
//
{ // initialize channel "brakePoint_53"
  hl3::byteSeq_t &byteSeq =
    hl3::byteSeq_t::create(sizeof(AnalogReal), sizeof(AnalogReal));
  AnalogReal &value = *(AnalogReal*)byteSeq.getValue();
  /* begin: initialization of value */
  value = 0;
  /* end: initialization of value */
  Channel::brakePoint_53->putInit(byteSeq, visSet);
  hl3::byteSeq_t::destroy(byteSeq);
}
/* ... */
{ // initialize channel "brakePoint_63"
  hl3::byteSeq_t &byteSeq =
    hl3::byteSeq_t::create(sizeof(AnalogReal), sizeof(AnalogReal));
  AnalogReal &value = *(AnalogReal*)byteSeq.getValue();
  /* begin: initialization of value */
  value = 0;
  /* end: initialization of value */
  Channel::brakePoint_63->putInit(byteSeq, visSet);
  hl3::byteSeq_t::destroy(byteSeq);
}
/* ... */

```

### D.1.2 Definition of Abstract Machines

The abstract machines  $m \in am(c_{tc})$  are also represented by C++ objects. Abstract machines internally encode the static structure of a hierarchic state-machine, as it is provided by  $mtree_{AIN}(ain_{Am}(m))$ , as a basis for their behavior (see section 5.6.1). For the chosen example, the internal structure corresponds to the top-level mode `brakePointControl` (Fig. D.1) of `BrakePointController`.

Corresponding to the algebraic and flow constraints of `brakePointControl`, HL<sup>3</sup> flows are created, as defined by  $flow(c_{tc})$ . Each HL<sup>3</sup> is represented by two objects – one object of class `Flow` which acts as the HL<sup>3</sup> flow itself, and one object of class `FlowConstraint`, which represents the association of the flow with its abstract machine. This represents the dependency  $am_{flow}(c_{tc})$ .

In the same fashion, the HL<sup>3</sup> transitions  $trans(c_{tc})$  are represented as `Transition` objects (along with a separate object that encodes the `Action`). Instances of class `ModeTransition` define the respective dependency  $am_{trans}(c_{tc})$  between abstract machines and HL<sup>3</sup> transitions.

The mapping  $prg_{subj}(c_{tc})$  of HL<sup>3</sup> flows and HL<sup>3</sup> transitions to the programs that implement the respective operation *integrate* or *action* is given by a reference to the specific C++ operation that encodes the corresponding expression. For the definition of these C++ operations see section D.2.

`Guard` and `InvariantConstraint` objects are instantiated to represent the expressions that are used internally during the abstract machine's execution,<sup>1</sup> as

<sup>1</sup>The class `Trigger` exists for trigger expressions, but no instances exist for the example `brakePointControl`.

well as objects of classes `Mode` and `ControlPoint` for the static structure.

```

/*****
/* BrakePointController */
*****/

/* abstract machine */
AbstractMachine am_TheBrakePointController_60x820e208();
const hl3::amId_t id_am_TheBrakePointController_60x820e208
  = am_TheBrakePointController_60x820e208.getId();
/* top-level mode: brakePointControl */
Mode mode_brakePointControl_150x8549700(0, 2, 2, 6, 0, 5, 4);
/* control points of brakePointControl */
ControlPoint cp_34(mode_brakePointControl_150x8549700, DEFAULT, 2, ENTRY);
ControlPoint cp_35(mode_brakePointControl_150x8549700, DEFAULT, 0, EXIT);
/* flows of brakePointControl: */
// [ flow: forall c in [0..CROSSING_COUNT-1] := {
//           t_closedTooLong[c]' := -1
//         } ]
Flow flow_expression53(1, &Flow::expression53);
FlowConstraint fc_expression53(flow_expression53);
mode_brakePointControl_150x8549700.addFlow(fc_expression53);
// [ alge: algeBrakePoint ]
Flow flow_expression54(1, &Flow::expression54);
FlowConstraint fc_expression54(flow_expression54);
mode_brakePointControl_150x8549700.addFlow(fc_expression54);
// [ alge: s.brake := (0-(c.v*v)^2)/(2*GC.a_min) ]
Flow flow_expression55(1, &Flow::expression55);
FlowConstraint fc_expression55(flow_expression55);
mode_brakePointControl_150x8549700.addFlow(fc_expression55);
// [ alge: t.brake := (0-(c.v*v))/GC.a_min ]
Flow flow_expression56(1, &Flow::expression56);
FlowConstraint fc_expression56(flow_expression56);
mode_brakePointControl_150x8549700.addFlow(fc_expression56);
// [ alge: algePosition1 ]
Flow flow_expression57(1, &Flow::expression57);
FlowConstraint fc_expression57(flow_expression57);
mode_brakePointControl_150x8549700.addFlow(fc_expression57);
// [ alge: algePosition2 ]
Flow flow_expression58(1, &Flow::expression58);
FlowConstraint fc_expression58(flow_expression58);
mode_brakePointControl_150x8549700.addFlow(fc_expression58);
/* no invariant constraints for brakePointControl */

/* submode of brakePointControl: brakingNotRequired */
Mode mode_BrakingNotRequired_160x8549778
  (&mode_brakePointControl_150x8549700, 2, 0, 0, 1, 0, 1);
/* control points of brakingNotRequired */
ControlPoint cp_30(mode_BrakingNotRequired_160x8549778, DEFAULT, 0, ENTRY);
ControlPoint cp_31(mode_BrakingNotRequired_160x8549778, DEFAULT, 1, EXIT);
/* no flows for brakingNotRequired */
/* invariant constraints of brakingNotRequired */
// [ inv: condBrakingNotRequired ]
InvariantConstraint inv_expression41(&InvariantConstraint::expression41);
mode_BrakingNotRequired_160x8549778.addInvariant(inv_expression41);
/* no submodes for brakingNotRequired */
/* no transitions for brakingNotRequired */

/* submode of brakePointControl: brakingRequired */
Mode mode_BrakingRequired_170x854f180
  (&mode_brakePointControl_150x8549700, 2, 0, 0, 2, 0, 2);
/* control points of brakingRequired */
ControlPoint cp_32(mode_BrakingRequired_170x854f180, DEFAULT, 0, ENTRY);

```

```

ControlPoint cp_33(mode_BrakingRequired_170x854f180, DEFAULT, 2, EXIT);
/* no flows for brakingRequired */
/* invariant constraints of brakingRequired */
// [ inv: condBrakingRequired ]
InvariantConstraint inv_expression42(&InvariantConstraint::expression42);
mode_BrakingRequired_170x854f180.addInvariant(inv_expression42);
// [ inv: ! condAdjustCrossingSafe ]
InvariantConstraint inv_expression43(&InvariantConstraint::expression43);
mode_BrakingRequired_170x854f180.addInvariant(inv_expression43);
/* no submodes for brakingRequired */
/* no transitions for brakingRequired */

/* transitions of brakePointControl */
//
// (init transition) brakePointControl —> BrakingNotRequired
// (empty action)
Action a_trans17(&Action::trans17ReadSet, &Action::trans17WriteSet,
                &Action::trans17);
Transition t_trans17(a_trans17, id_am_TheBrakePointController_60x820e208);
// (implicitly) [ true ]
Guard grd_expression44(&Guard::expression44);
ModeTransition mt_trans17(cp_30, grd_expression44, t_trans17);
cp_34.addTransOut(mt_trans17);
//
// (init transition) brakePointControl —> BrakingNotRequired
// (empty action)
Action a_trans18(&Action::trans18ReadSet, &Action::trans18WriteSet,
                &Action::trans18);
Transition t_trans18(a_trans18, id_am_TheBrakePointController_60x820e208);
// (implicitly) [ true ]
Guard grd_expression45(&Guard::expression45);
ModeTransition mt_trans18(cp_32, grd_expression45, t_trans18);
cp_34.addTransOut(mt_trans18);
//
// BrakingNotRequired —> BrakingRequired
// brakingRequired := true
Action a_trans19(&Action::trans19ReadSet, &Action::trans19WriteSet,
                &Action::trans19);
Transition t_trans19(a_trans19, id_am_TheBrakePointController_60x820e208);
// [ condBrakingRequired ]
Guard grd_expression46(&Guard::expression46);
ModeTransition mt_trans19(cp_32, grd_expression46, t_trans19);
cp_31.addTransOut(mt_trans19);
//
// BrakingRequired —> BrakingRequired
// cid := crossingSafeToAdjust ; crossingSafe[cid] := false
Action a_trans20(&Action::trans20ReadSet, &Action::trans20WriteSet,
                &Action::trans20);
Transition t_trans20(a_trans20, id_am_TheBrakePointController_60x820e208);
// [ condAdjustCrossingSafe ]
Guard grd_expression48(&Guard::expression48);
ModeTransition mt_trans20(cp_32, grd_expression48, t_trans20);
cp_33.addTransOut(mt_trans20);
//
// BrakingRequired —> BrakingNotRequired
// brakingRequired := false
Action a_trans21(&Action::trans21ReadSet, &Action::trans21WriteSet,
                &Action::trans21);
Transition t_trans21(a_trans21, id_am_TheBrakePointController_60x820e208);
// [ condBrakingNotRequired ]
Guard grd_expression51(&Guard::expression51);
ModeTransition mt_trans21(cp_30, grd_expression51, t_trans21);

```

```

cp_33.addTransOut(mt_trans21);

/* assignment of top-level mode brakePointControl */
/* to abstract machine BrakePointController */
am_TheBrakePointController_60x820e208.setTopLevelMode
(mode_brakePointControl_150x8549700);

```

### D.1.3 Instantiation of Selector and Scheduler

Finally, the pre-defined HybridUML selector  $sel(c_{tc})$ , as well as the HL<sup>3</sup> scheduler  $sched(c_{tc})$  are instantiated. Afterwards, execution is started.

```

/* ... */

/*****
/* HybridUML Selector */
*****/
Selector selector();

/*****
/* Scheduler */
*****/
hl3::Scheduler &scheduler =
    hl3::Scheduler::createScheduler(SYS_PERIOD, selector);

scheduler.run();
}

```

Note that some components of  $c_{tc}$  are not explicitly given: (1) Since there are no interface modules for the simulation semantics of HybridUML, the set of interface modules  $ifm(c_{tc})$  itself, as well as visibilities  $vis_{ifm}(c_{tc})$  and period factors  $period_{ifm,poll}(c_{tc})$ ,  $period_{ifm,tmit}(c_{tc})$  do not occur. (2) The current implementation of the HL<sup>3</sup> scheduler only supports a single light weight process, therefore the set of light weight processes  $lwp(c_{tc})$  is given implicitly, as well as the mapping of abstract machines and the selector to light weight processes  $lwp_{subj}(c_{tc})$ . (3) The visibilities of the execution results of HL<sup>3</sup> flows and HL<sup>3</sup> transitions are always unrestricted for HybridUML, therefore  $vis_{flow}(c_{tc})$  and  $vis_{trans}(c_{tc})$  are fixed for all flows and transitions, and are not presented here. (4) The set of ports  $selport(c_{tc})$  that are accessible by the selector is realized by object references, but is not presented. (5) The system period  $\delta_{period}(c_{tc})$  (given as SYS\_PERIOD for the instantiation of the scheduler) has to be adjusted manually, because it cannot be generated from HybridUML specifications. The flow periods  $period_{flow}(c_{tc})$  also cannot be determined automatically and have a default value. (6) The mapping  $subject_{port}(c_{tc})$  of ports to the subjects which access the port is represented implicitly: ports are accessed by program code within C++ operations that encode expressions. (7) The same holds for local variables, such that  $subject_{var}(c_{tc})$ , as well as set of local variables  $var(c_{tc})$  itself, is reflected by the assignment of local variables to their containing C++ operations. Note that local C++ variables of course are declared *within* the corresponding operation, rather than separately.

## D.2 Created Code for Expression Nodes

In this section, all C++ operations are given that implement the expressions of mode `brakePointControl`. They correspond to the program definitions which result from the creation rules of section 5.5. The C++ variants given here are generated by the implementation of transformation  $\Phi_{HUMML}$ . For an explanation of how these programs are generated, consult section 5.5.

### D.2.1 Flow::expression53

```

void Flow::expression53 (const hl3::VisibilitySet &_vis) {
    //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
    // forall c in [0..CROSSING_COUNT-1] := { t.closedTooLong[c] := -1 } //
    //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

    // Initialization of Local Variables from Channels
    const Integer *_array_CROSSING_COUNT[] = {
        &Port::CROSSING_COUNT_280x853fce8->getBuffered()
    };
    hl3::PArrayWrapper<const Integer, index_t, 1>
        CROSSING_COUNT(_array_CROSSING_COUNT);
    const AnalogReal *_array_t_closedTooLong[] = {
        &Port::t_closedTooLong_330x853fd38->getBuffered(),
        &Port::t_closedTooLong_340x853fd48->getBuffered(),
        &Port::t_closedTooLong_350x853fd58->getBuffered()
    };
    hl3::PArrayWrapper<const AnalogReal, index_t, 3>
        t_closedTooLong(_array_t_closedTooLong);
    AnalogReal *_w__array_t_closedTooLong[] = {
        &Port::t_closedTooLong_330x853fd38->getDataBuffer(),
        &Port::t_closedTooLong_340x853fd48->getDataBuffer(),
        &Port::t_closedTooLong_350x853fd58->getDataBuffer()
    };
    hl3::PArrayWrapper<AnalogReal, index_t, 3>
        _w_t_closedTooLong(_w__array_t_closedTooLong);
    hl3::FixedArray<hl3::TimeTick, index_t, 3> _tck_t_closedTooLong;
    _tck_t_closedTooLong[0] = Port::t_closedTooLong_330x853fd38->getTime();
    _tck_t_closedTooLong[1] = Port::t_closedTooLong_340x853fd48->getTime();
    _tck_t_closedTooLong[2] = Port::t_closedTooLong_350x853fd58->getTime();

    // Assignment Interpretation
    const hl3::VisibilitySet &_visOrig = _vis;
    hl3::VisibilitySet _vis1(1,1);
    for (hl3::visibilityAttributeCount_t visIndex = 0;
        visIndex < _visOrig.size(); ++visIndex) {
        hl3::VisibilitySet &_vis = _vis1;
        _vis1[0] = _visOrig[visIndex];
        const hl3::TimeTick &_newtck = _vis1[0]->left();
        {
            hl3::FixedArray<Integer, index_t, 1> c;
            for (c[0] = 0; c[0] <= (CROSSING_COUNT[0] - 1); ++c[0]) {
                _w_t_closedTooLong[c[0]] =
                    (-1) * (_newtck.t0_s() - _tck_t_closedTooLong[c[0]].t0_s())
                    + t_closedTooLong[c[0]];
            }
        }
    }
    // Publication of Local Variable Data to Channels
    Port::t_closedTooLong_330x853fd38->put(_vis);
    Port::t_closedTooLong_340x853fd48->put(_vis);
    Port::t_closedTooLong_350x853fd58->put(_vis);
}

```

```

}
}

```

### D.2.2 Flow::expression54

```

void Flow::expression54 (const hl3::VisibilitySet &_vis) {
////////////////////////////////////////////////////////////////////
// algeBrakePoint //
// //
// forall i in [0..VTP_COUNT-1] := { //
//   brakePoint[i] := RA.vtp[i].x //
//   - (RA.vtp[i].v^2-(c.v*v)^2)/(2*GC.a_min) - s_safe } //
////////////////////////////////////////////////////////////////////

// Initialization of Local Variables from Channels
const RouteAtlas *_array_RA [] = {
  &Port::RA_310x853fd18->getBuffered()
};
hl3::PArrayWrapper<const RouteAtlas, index_t, 1> RA(_array_RA);
const Integer *_array_VTP_COUNT [] = {
  &Port::VTP_COUNT_290x853fcf8->getBuffered()
};
hl3::PArrayWrapper<const Integer, index_t, 1> VTP_COUNT(_array_VTP_COUNT);
const GlobalConstants *_array_GC [] = {
  &Port::GC_300x853fd08->getBuffered()
};
hl3::PArrayWrapper<const GlobalConstants, index_t, 1> GC(_array_GC);
const AnalogReal *_array_v [] = {
  &Port::v_380x853fe90->getBuffered()
};
hl3::PArrayWrapper<const AnalogReal, index_t, 1> v(_array_v);
const Real *_array_c_v [] = {
  &Port::c_v_730x853fdb8->getBuffered()
};
hl3::PArrayWrapper<const Real, index_t, 1> c_v(_array_c_v);
const Real *_array_s_safe [] = {
  &Port::s_safe_740x853fdc8->getBuffered()
};
hl3::PArrayWrapper<const Real, index_t, 1> s_safe(_array_s_safe);
AnalogReal *_w__array_brakePoint [] = {
  &Port::brakePoint_530x853ff80->getDataBuffer(),
  &Port::brakePoint_540x853ff90->getDataBuffer(),
  &Port::brakePoint_550x853ffa0->getDataBuffer(),
  &Port::brakePoint_560x853ffb0->getDataBuffer(),
  &Port::brakePoint_570x853ffc0->getDataBuffer(),
  &Port::brakePoint_580x853ffd0->getDataBuffer(),
  &Port::brakePoint_590x853ffe0->getDataBuffer(),
  &Port::brakePoint_600x853fff0->getDataBuffer(),
  &Port::brakePoint_610x8540000->getDataBuffer(),
  &Port::brakePoint_620x8540010->getDataBuffer(),
  &Port::brakePoint_630x8540020->getDataBuffer()
};
hl3::PArrayWrapper<AnalogReal, index_t, 11>
  _w_brakePoint(_w__array_brakePoint);

// Assignment Interpretation
{
  hl3::FixedArray<Integer, index_t, 1> i;
  for (i[0] = 0; i[0] <= (VTP_COUNT[0] - 1); ++i[0]) {
    _w_brakePoint[i[0]] =
      ((RA[0].vtp[i[0]].x[0] - ((powf((float)RA[0].vtp[i[0]].v[0], (float)2)
        - powf((float)(c_v[0] * v[0]), (float)2))

```

```

    / (2 * GC[0].a_min[0])) - s_safe[0]);
}
}

// Publication of Local Variable Data to Channels
Port::brakePoint_530x853ff80->put(_vis);
Port::brakePoint_540x853ff90->put(_vis);
Port::brakePoint_550x853ffa0->put(_vis);
Port::brakePoint_560x853ffb0->put(_vis);
Port::brakePoint_570x853ffc0->put(_vis);
Port::brakePoint_580x853ffd0->put(_vis);
Port::brakePoint_590x853ffe0->put(_vis);
Port::brakePoint_600x853fff0->put(_vis);
Port::brakePoint_610x8540000->put(_vis);
Port::brakePoint_620x8540010->put(_vis);
Port::brakePoint_630x8540020->put(_vis);

}

```

### D.2.3 Flow::expression55

```

void Flow::expression55 (const hl3::VisibilitySet &_vis) {
    //////////////////////////////////////
    // s_brake := (0-(c*v*v)^2)/(2*GC.a_min) //
    //////////////////////////////////////

    // Initialization of Local Variables from Channels
    const GlobalConstants *_array_GC[] = {
        &Port::GC_300x853fd08->getBuffered()
    };
    hl3::PArrayWrapper<const GlobalConstants, index_t, 1> GC(_array_GC);
    const AnalogReal *_array_v[] = {
        &Port::v_380x853fe90->getBuffered()
    };
    hl3::PArrayWrapper<const AnalogReal, index_t, 1> v(_array_v);
    const Real *_array_c_v[] = {
        &Port::c_v_730x853fdb8->getBuffered()
    };
    hl3::PArrayWrapper<const Real, index_t, 1> c_v(_array_c_v);
    AnalogReal *_w__array_s_brake[] = {
        &Port::s_brake_640x8540030->getDataBuffer()
    };
    hl3::PArrayWrapper<AnalogReal, index_t, 1> _w_s_brake(_w__array_s_brake);

    // Assignment Interpretation
    _w_s_brake[0] =
        ((0 - powf((float)(c_v[0] * v[0]), (float)2)) / (2 * GC[0].a_min[0]));

    // Publication of Local Variable Data to Channels
    Port::s_brake_640x8540030->put(_vis);

}

```

### D.2.4 Flow::expression56

```

void Flow::expression56 (const hl3::VisibilitySet &_vis) {
    //////////////////////////////////////
    // t_brake := (0-(c*v*v))/GC.a_min //
    //////////////////////////////////////

    // Initialization of Local Variables from Channels
    const GlobalConstants *_array_GC[] = {

```

```

    &Port::GC_300x853fd08->getBuffered()
};
hl3::PArrayWrapper<const GlobalConstants,index_t,1> GC(_array_GC);
const AnalogReal *_array_v[] = {
    &Port::v_380x853fe90->getBuffered()
};
hl3::PArrayWrapper<const AnalogReal,index_t,1> v(_array_v);
const Real *_array_c_v[] = {
    &Port::c_v_730x853fdb8->getBuffered()
};
hl3::PArrayWrapper<const Real,index_t,1> c_v(_array_c_v);
AnalogReal *_w__array_t_brake[] = {
    &Port::t_brake_650x8540040->getDataBuffer()
};
hl3::PArrayWrapper<AnalogReal,index_t,1> _w_t_brake(_w__array_t_brake);

// Assignment Interpretation
_w_t_brake[0] = ((0 - (c_v[0] * v[0])) / GC[0].a_min[0]);

// Publication of Local Variable Data to Channels
Port::t_brake_650x8540040->put(_vis);
}

```

### D.2.5 Flow::expression57

```

void Flow::expression57 (const hl3::VisibilitySet &_vis) {
    ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
    // algePosition1
    //
    // forall c in [0..CROSSING_COUNT-1] := {
    //   x_closedTooLong_1[c] := x + GC.a_min/2 * t_closedTooLong[c]^2 //
    //   + (c*v*v)*t_closedTooLong[c] - s_safe }
    ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

    // Initialization of Local Variables from Channels
    const Integer *_array_CROSSING_COUNT[] = {
        &Port::CROSSING_COUNT_280x853fce8->getBuffered()
    };
    hl3::PArrayWrapper<const Integer,index_t,1>
        CROSSING_COUNT(_array_CROSSING_COUNT);
    const GlobalConstants *_array_GC[] = {
        &Port::GC_300x853fd08->getBuffered()
    };
    hl3::PArrayWrapper<const GlobalConstants,index_t,1> GC(_array_GC);
    const AnalogReal *_array_x[] = {
        &Port::x_370x853fd78->getBuffered()
    };
    hl3::PArrayWrapper<const AnalogReal,index_t,1> x(_array_x);
    const AnalogReal *_array_v[] = {
        &Port::v_380x853fe90->getBuffered()
    };
    hl3::PArrayWrapper<const AnalogReal,index_t,1> v(_array_v);
    const AnalogReal *_array_t_closedTooLong[] = {
        &Port::t_closedTooLong_330x853fd38->getBuffered(),
        &Port::t_closedTooLong_340x853fd48->getBuffered(),
        &Port::t_closedTooLong_350x853fd58->getBuffered()
    };
    hl3::PArrayWrapper<const AnalogReal,index_t,3>
        t_closedTooLong(_array_t_closedTooLong);
    const Real *_array_c_v[] = {
        &Port::c_v_730x853fdb8->getBuffered()
    };
};

```

```

h13::PArrayWrapper<const Real,index_t,1> c_v(_array_c_v);
const Real *_array_s_safe[] = {
    &Port::s_safe_740x853fdc8->getBuffered()
};
h13::PArrayWrapper<const Real,index_t,1> s_safe(_array_s_safe);
AnalogReal *_w__array_x_closedTooLong_1[] = {
    &Port::x_closedTooLong_1_660x8540050->getDataBuffer(),
    &Port::x_closedTooLong_1_670x8540060->getDataBuffer(),
    &Port::x_closedTooLong_1_680x8540070->getDataBuffer()
};
h13::PArrayWrapper<AnalogReal,index_t,3>
    _w_x_closedTooLong_1(_w__array_x_closedTooLong_1);

// Assignment Interpretation
{
    h13::FixedArray<Integer,index_t,1> c;
    for (c[0] = 0; c[0] <= (CROSSING_COUNT[0] - 1); ++c[0]) {
        _w_x_closedTooLong_1[c[0]] =
            (((x[0] + ((GC[0].a_min[0] / 2)
                * powf((float)t_closedTooLong[c[0]],(float)2)))
            + ((c_v[0] * v[0]) * t_closedTooLong[c[0]])) - s_safe[0]);
    }
}

// Publication of Local Variable Data to Channels
Port::x_closedTooLong_1_660x8540050->put(_vis);
Port::x_closedTooLong_1_670x8540060->put(_vis);
Port::x_closedTooLong_1_680x8540070->put(_vis);
}

```

## D.2.6 Flow::expression58

```

void Flow::expression58 (const h13::VisibilitySet &_vis) {
    ////////////////////////////////////////
    // algePosition2 //
    // //
    // forall c in [0..CROSSING_COUNT-1] := { //
    //     x_closedTooLong_2[c] := x + s_brake + //
    //     (t_closedTooLong[c]-t_brake) * GC.v_pass - s_safe } //
    // ////////////////////////////////////////

    // Initialization of Local Variables from Channels
    const Integer *_array_CROSSING_COUNT[] = {
        &Port::CROSSING_COUNT_280x853fce8->getBuffered()
    };
    h13::PArrayWrapper<const Integer,index_t,1>
        CROSSING_COUNT(_array_CROSSING_COUNT);
    const GlobalConstants *_array_GC[] = {
        &Port::GC_300x853fd08->getBuffered()
    };
    h13::PArrayWrapper<const GlobalConstants,index_t,1> GC(_array_GC);
    const AnalogReal *_array_x[] = {
        &Port::x_370x853fd78->getBuffered()
    };
    h13::PArrayWrapper<const AnalogReal,index_t,1> x(_array_x);
    const AnalogReal *_array_t_closedTooLong[] = {
        &Port::t_closedTooLong_330x853fd38->getBuffered(),
        &Port::t_closedTooLong_340x853fd48->getBuffered(),
        &Port::t_closedTooLong_350x853fd58->getBuffered()
    };
    h13::PArrayWrapper<const AnalogReal,index_t,3>
        t_closedTooLong(_array_t_closedTooLong);
}

```

```

    const AnalogReal *_array_s_brake[] = {
        &Port::s_brake_640x8540030->getBuffered()
    };
    hl3::PArrayWrapper<const AnalogReal, index_t, 1> s_brake(_array_s_brake);
    const AnalogReal *_array_t_brake[] = {
        &Port::t_brake_650x8540040->getBuffered()
    };
    hl3::PArrayWrapper<const AnalogReal, index_t, 1> t_brake(_array_t_brake);
    const Real *_array_s_safe[] = {
        &Port::s_safe_740x853fdc8->getBuffered()
    };
    hl3::PArrayWrapper<const Real, index_t, 1> s_safe(_array_s_safe);
    AnalogReal *_w__array_x_closedTooLong_2[] = {
        &Port::x_closedTooLong_2_690x8540080->getDataBuffer(),
        &Port::x_closedTooLong_2_700x853fd88->getDataBuffer(),
        &Port::x_closedTooLong_2_710x853fd98->getDataBuffer()
    };
    hl3::PArrayWrapper<AnalogReal, index_t, 3>
        _w_x_closedTooLong_2(_w__array_x_closedTooLong_2);

    // Assignment Interpretation
    {
        hl3::FixedArray<Integer, index_t, 1> c;
        for (c[0] = 0; c[0] <= (CROSSING_COUNT[0] - 1); ++c[0]) {
            _w_x_closedTooLong_2[c[0]] =
                ((x[0] + s_brake[0]) + ((t_closedTooLong[c[0]] - t_brake[0])
                    * GC[0].v_pass[0])) - s_safe[0];
        }
    }

    // Publication of Local Variable Data to Channels
    Port::x_closedTooLong_2_690x8540080->put(_vis);
    Port::x_closedTooLong_2_700x853fd88->put(_vis);
    Port::x_closedTooLong_2_710x853fd98->put(_vis);
}

```

### D.2.7 Action::trans17

```

void Action::trans17 (const hl3::VisibilitySet &_vis) {
    // (empty action)
}

```

### D.2.8 Action::trans18

```

void Action::trans18 (const hl3::VisibilitySet &_vis) {
    // (empty action)
}

```

### D.2.9 Action::trans19

```

void Action::trans19 (const hl3::VisibilitySet &_vis) {
    ////////////////////////////////////
    // brakingRequired := true //
    ////////////////////////////////////

    // Initialization of Local Variables from Channels
    Boolean *_w__array_brakingRequired[] = {
        &Port::brakingRequired_360x853fd68->getDataBuffer()
    };
    hl3::PArrayWrapper<Boolean, index_t, 1>
        _w_brakingRequired(_w__array_brakingRequired);
}

```

```

// Assignment Interpretation
_w_brakingRequired[0] = true;

// Publication of Local Variable Data to Channels
Port::brakingRequired_360x853fd68->put(_vis);
}

```

## D.2.10 Action::trans20

```

void Action::trans20 (const h13::VisibilitySet &_vis) {
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// cid :=in crossingSafeToAdjust ; crossingSafe[cid] := false //
// // //
// crossingSafeToAdjust: //
// // //
// c in [0..CROSSING_COUNT-1] : { //
// crossingSafe[c] //
// && ((x_closedTooLong_1[c] <= RA.cr[c].x_end + GC.trainLength //
// && t_closedTooLong[c] <= t_brake) //
// || (x_closedTooLong_2[c] <= RA.cr[c].x_end + GC.trainLength //
// && t_closedTooLong[c] > t_brake)) //
// } //
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

// Initialization of Local Variables from Channels
const RouteAtlas *_array_RA [] = {
&Port::RA_310x853fd18->getBuffered()
};
h13::PArrayWrapper<const RouteAtlas, index_t, 1> RA(_array_RA);
const Integer *_array_CROSSING_COUNT [] = {
&Port::CROSSING_COUNT_280x853fce8->getBuffered()
};
h13::PArrayWrapper<const Integer, index_t, 1>
CROSSING_COUNT(_array_CROSSING_COUNT);
const GlobalConstants *_array_GC [] = {
&Port::GC_300x853fd08->getBuffered()
};
h13::PArrayWrapper<const GlobalConstants, index_t, 1> GC(_array_GC);
const AnalogReal *_array_t_closedTooLong [] = {
&Port::t_closedTooLong_330x853fd38->getBuffered(),
&Port::t_closedTooLong_340x853fd48->getBuffered(),
&Port::t_closedTooLong_350x853fd58->getBuffered()
};
h13::PArrayWrapper<const AnalogReal, index_t, 3>
t_closedTooLong(_array_t_closedTooLong);
const Boolean *_array_crossingSafe [] = {
&Port::crossingSafe_500x853ff50->getBuffered(),
&Port::crossingSafe_510x853ff60->getBuffered(),
&Port::crossingSafe_520x853ff70->getBuffered()
};
h13::PArrayWrapper<const Boolean, index_t, 3>
crossingSafe(_array_crossingSafe);
const AnalogReal *_array_t_brake [] = {
&Port::t_brake_650x8540040->getBuffered()
};
h13::PArrayWrapper<const AnalogReal, index_t, 1> t_brake(_array_t_brake);
const AnalogReal *_array_x_closedTooLong_1 [] = {
&Port::x_closedTooLong_1_660x8540050->getBuffered(),
&Port::x_closedTooLong_1_670x8540060->getBuffered(),
&Port::x_closedTooLong_1_680x8540070->getBuffered()
};
h13::PArrayWrapper<const AnalogReal, index_t, 3>

```

```

    x_closedTooLong_1(_array_x_closedTooLong_1);
    const AnalogReal *_array_x_closedTooLong_2[] = {
        &Port::x_closedTooLong_2_690x8540080->getBuffered(),
        &Port::x_closedTooLong_2_700x853fd88->getBuffered(),
        &Port::x_closedTooLong_2_710x853fd98->getBuffered()
    };
    hl3::PArrayWrapper<const AnalogReal, index_t, 3>
    x_closedTooLong_2(_array_x_closedTooLong_2);
    Integer *_w__array_cid[] = {
        &Port::cid_720x853fda8->getDataBuffer()
    };
    hl3::PArrayWrapper<Integer, index_t, 1> _w_cid(_w__array_cid);
    const Integer *_array_cid[] = {
        &Port::cid_720x853fda8->getBuffered()
    };
    hl3::PArrayWrapper<const Integer, index_t, 1> cid(_array_cid);
    Boolean *_w__array_crossingSafe[] = {
        &Port::crossingSafe_500x853ff50->getDataBuffer(),
        &Port::crossingSafe_510x853ff60->getDataBuffer(),
        &Port::crossingSafe_520x853ff70->getDataBuffer()
    };
    hl3::PArrayWrapper<Boolean, index_t, 3>
    _w_crossingSafe(_w__array_crossingSafe);

// Assignment Interpretation
{
    const Integer _MAX_HIT_COUNT = 117;
    Integer _hitCount = 0;
    hl3::FixedArray<Integer, index_t, _MAX_HIT_COUNT> _hits;
    hl3::FixedArray<Integer, index_t, 1> c;
    for (c[0] = 0;
        (c[0] <= (CROSSING_COUNT[0] - 1)) && (_hitCount < _MAX_HIT_COUNT);
        ++c[0]) {
        if ((crossingSafe[c[0]] &&
            (((x_closedTooLong_1[c[0]] <= (RA[0].cr[c[0]].x_end[0]
                + GC[0].trainLength[0]))
            && (t_closedTooLong[c[0]] <= t_brake[0]))
            || ((x_closedTooLong_2[c[0]] <= (RA[0].cr[c[0]].x_end[0]
                + GC[0].trainLength[0]))
            && (t_closedTooLong[c[0]] > t_brake[0])))) {
            _hits[_hitCount++] = c[0];
        }
    }
    if (_hitCount > 0) {
        _w_cid[0] = _hits[random() % _hitCount];
    }
}
_w_crossingSafe[cid[0]] = false;

// Publication of Local Variable Data to Channels
Port::cid_720x853fda8->put(_vis);
Port::crossingSafe_500x853ff50->put(_vis);
Port::crossingSafe_510x853ff60->put(_vis);
Port::crossingSafe_520x853ff70->put(_vis);
}

```

### D.2.11 Action::trans21

```

void Action::trans21 (const hl3::VisibilitySet &_vis) {
    //////////////////////////////////////
    // brakingRequired := false //
    //////////////////////////////////////
}

```

```

// Initialization of Local Variables from Channels
Boolean *_w__array_brakingRequired[] = {
    &Port::brakingRequired_360x853fd68->getDataBuffer()
};
hl3::PArrayWrapper<Boolean, index_t, 1>
    _w_brakingRequired(_w__array_brakingRequired);

// Assignment Interpretation
_w_brakingRequired[0] = false;

// Publication of Local Variable Data to Channels
Port::brakingRequired_360x853fd68->put(_vis);
}

```

### D.2.12 InvariantConstraint::expression41

```

bool InvariantConstraint::expression41 () {
    ////////////////////////////////////////////////////////////////////
    // condBrakingNotRequired //
    // // //
    // forall i in [0..VTP_COUNT-1] : { !( //
    //     brakePoint[i] <= x && RA.vtp[i].v < v && RA.vtp[i].x > x //
    //     && (vtpActive[i] || //
    //         RA.vtp[i].type == VTP_TYPE::CROSSING //
    //         && ((x_closedTooLong_1[RA.vtp[i].crld] //
    //             <= RA.cr[RA.vtp[i].crld].x_end + GC.trainLength //
    //             && t_closedTooLong[RA.vtp[i].crld] <= t_brake) //
    //         || (x_closedTooLong_2[RA.vtp[i].crld] //
    //             <= RA.cr[RA.vtp[i].crld].x_end + GC.trainLength //
    //             && t_closedTooLong[RA.vtp[i].crld] > t_brake))) //
    //     )} //
    ////////////////////////////////////////////////////////////////////

    // Initialization of Local Variables from Channels
    const RouteAtlas *_array_RA[] = {
        &Port::RA_310x853fd18->getBuffered()
    };
    hl3::PArrayWrapper<const RouteAtlas, index_t, 1> RA(_array_RA);
    const Integer *_array_VTP_COUNT[] = {
        &Port::VTP_COUNT_290x853fcf8->getBuffered()
    };
    hl3::PArrayWrapper<const Integer, index_t, 1> VTP_COUNT(_array_VTP_COUNT);
    const GlobalConstants *_array_GC[] = {
        &Port::GC_300x853fd08->getBuffered()
    };
    hl3::PArrayWrapper<const GlobalConstants, index_t, 1> GC(_array_GC);
    const AnalogReal *_array_brakePoint[] = {
        &Port::brakePoint_530x853ff80->getBuffered(),
        &Port::brakePoint_540x853ff90->getBuffered(),
        &Port::brakePoint_550x853ffa0->getBuffered(),
        &Port::brakePoint_560x853ffb0->getBuffered(),
        &Port::brakePoint_570x853ffc0->getBuffered(),
        &Port::brakePoint_580x853ffd0->getBuffered(),
        &Port::brakePoint_590x853ffe0->getBuffered(),
        &Port::brakePoint_600x853fff0->getBuffered(),
        &Port::brakePoint_610x8540000->getBuffered(),
        &Port::brakePoint_620x8540010->getBuffered(),
        &Port::brakePoint_630x8540020->getBuffered()
    };
    hl3::PArrayWrapper<const AnalogReal, index_t, 11>
        brakePoint(_array_brakePoint);
}

```

```

const AnalogReal *_array_x[] = {
    &Port::x_370x853fd78->getBuffered()
};
hl3::PArrayWrapper<const AnalogReal, index_t, 1> x(_array_x);
const AnalogReal *_array_v[] = {
    &Port::v_380x853fe90->getBuffered()
};
hl3::PArrayWrapper<const AnalogReal, index_t, 1> v(_array_v);
const AnalogReal *_array_t_closedTooLong[] = {
    &Port::t_closedTooLong_330x853fd38->getBuffered(),
    &Port::t_closedTooLong_340x853fd48->getBuffered(),
    &Port::t_closedTooLong_350x853fd58->getBuffered()
};
hl3::PArrayWrapper<const AnalogReal, index_t, 3>
t_closedTooLong(_array_t_closedTooLong);
const Boolean *_array_vtpActive[] = {
    &Port::vtpActive_390x853fea0->getBuffered(),
    &Port::vtpActive_400x853feb0->getBuffered(),
    &Port::vtpActive_410x853fec0->getBuffered(),
    &Port::vtpActive_420x853fed0->getBuffered(),
    &Port::vtpActive_430x853fee0->getBuffered(),
    &Port::vtpActive_440x853fef0->getBuffered(),
    &Port::vtpActive_450x853ff00->getBuffered(),
    &Port::vtpActive_460x853ff10->getBuffered(),
    &Port::vtpActive_470x853ff20->getBuffered(),
    &Port::vtpActive_480x853ff30->getBuffered(),
    &Port::vtpActive_490x853ff40->getBuffered()
};
hl3::PArrayWrapper<const Boolean, index_t, 11> vtpActive(_array_vtpActive);
const AnalogReal *_array_t_brake[] = {
    &Port::t_brake_650x8540040->getBuffered()
};
hl3::PArrayWrapper<const AnalogReal, index_t, 1> t_brake(_array_t_brake);
const AnalogReal *_array_x_closedTooLong_1[] = {
    &Port::x_closedTooLong_1_660x8540050->getBuffered(),
    &Port::x_closedTooLong_1_670x8540060->getBuffered(),
    &Port::x_closedTooLong_1_680x8540070->getBuffered()
};
hl3::PArrayWrapper<const AnalogReal, index_t, 3>
x_closedTooLong_1(_array_x_closedTooLong_1);
const AnalogReal *_array_x_closedTooLong_2[] = {
    &Port::x_closedTooLong_2_690x8540080->getBuffered(),
    &Port::x_closedTooLong_2_700x853fd88->getBuffered(),
    &Port::x_closedTooLong_2_710x853fd98->getBuffered()
};
hl3::PArrayWrapper<const AnalogReal, index_t, 3>
x_closedTooLong_2(_array_x_closedTooLong_2);

// Evaluation Interpretation
bool _result0x854ac08 = true;
{
    hl3::FixedArray<Integer, index_t, 1> i;
    for (i[0] = 0; (i[0] <= (VTP_COUNT[0] - 1)) && _result0x854ac08; ++i[0]) {
        _result0x854ac08 =
            (! (((brakePoint[i[0]] <= x[0]) && (RA[0].vtp[i[0]].v[0] < v[0]))
                && (RA[0].vtp[i[0]].x[0] > x[0]))
                && (vtpActive[i[0]]
                    || ((RA[0].vtp[i[0]].type[0] == VTP_TYPE_CROSSING)
                        && (((x_closedTooLong_1[RA[0].vtp[i[0]].crId[0]]
                            <= (RA[0].cr[RA[0].vtp[i[0]].crId[0]].x_end[0]
                                + GC[0].trainLength[0]))
                            && (t_closedTooLong[RA[0].vtp[i[0]].crId[0]]

```

```

        <= t_brake[0]))
    || ((x_closedTooLong_2[RA[0].vtp[i[0]].crId[0]]
        <= (RA[0].cr[RA[0].vtp[i[0]].crId[0]].x_end[0]
            + GC[0].trainLength[0]))
        && (t_closedTooLong[RA[0].vtp[i[0]].crId[0]]
            > t_brake[0])))))));
    }
}
return _result0x854ac08;
}

```

### D.2.13 InvariantConstraint::expression42

```

bool InvariantConstraint::expression42 () {
    ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
    // condBrakingRequired //
    // //
    // exists i in [0..VTP_COUNT-1] : { ( //
    //   brakePoint[i] <= x && RA.vtp[i].v < v && RA.vtp[i].x > x //
    //   && (vtpActive[i] || //
    //     RA.vtp[i].type == VTP_TYPE::CROSSING //
    //     && ((x_closedTooLong_1[RA.vtp[i].crId] //
    //         <= RA.cr[RA.vtp[i].crId].x_end + GC.trainLength //
    //         && t_closedTooLong[RA.vtp[i].crId] <= t_brake) //
    //     || (x_closedTooLong_2[RA.vtp[i].crId] //
    //         <= RA.cr[RA.vtp[i].crId].x_end + GC.trainLength //
    //         && t_closedTooLong[RA.vtp[i].crId] > t_brake))) //
    //   )} //
    ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

    // Initialization of Local Variables from Channels
    const RouteAtlas *_array_RA[] = {
        &Port::RA_310x853fd18->getBuffered()
    };
    hl3::PArrayWrapper<const RouteAtlas, index_t, 1> RA(_array_RA);
    const Integer *_array_VTP_COUNT[] = {
        &Port::VTP_COUNT_290x853fcf8->getBuffered()
    };
    hl3::PArrayWrapper<const Integer, index_t, 1> VTP_COUNT(_array_VTP_COUNT);
    const GlobalConstants *_array_GC[] = {
        &Port::GC_300x853fd08->getBuffered()
    };
    hl3::PArrayWrapper<const GlobalConstants, index_t, 1> GC(_array_GC);
    const AnalogReal *_array_brakePoint[] = {
        &Port::brakePoint_530x853ff80->getBuffered(),
        &Port::brakePoint_540x853ff90->getBuffered(),
        &Port::brakePoint_550x853ffa0->getBuffered(),
        &Port::brakePoint_560x853ffb0->getBuffered(),
        &Port::brakePoint_570x853ffc0->getBuffered(),
        &Port::brakePoint_580x853ffd0->getBuffered(),
        &Port::brakePoint_590x853ffe0->getBuffered(),
        &Port::brakePoint_600x853fff0->getBuffered(),
        &Port::brakePoint_610x8540000->getBuffered(),
        &Port::brakePoint_620x8540010->getBuffered(),
        &Port::brakePoint_630x8540020->getBuffered()
    };
    hl3::PArrayWrapper<const AnalogReal, index_t, 11>
        brakePoint(_array_brakePoint);
    const AnalogReal *_array_x[] = {
        &Port::x_370x853fd78->getBuffered()
    };
    hl3::PArrayWrapper<const AnalogReal, index_t, 1> x(_array_x);
}

```

```

const AnalogReal *_array_v[] = {
    &Port::v_380x853fe90->getBuffered()
};
hl3::PArrayWrapper<const AnalogReal, index_t, 1> v(_array_v);
const AnalogReal *_array_t_closedTooLong[] = {
    &Port::t_closedTooLong_330x853fd38->getBuffered(),
    &Port::t_closedTooLong_340x853fd48->getBuffered(),
    &Port::t_closedTooLong_350x853fd58->getBuffered()
};
hl3::PArrayWrapper<const AnalogReal, index_t, 3>
    t_closedTooLong(_array_t_closedTooLong);
const Boolean *_array_vtpActive[] = {
    &Port::vtpActive_390x853fea0->getBuffered(),
    &Port::vtpActive_400x853feb0->getBuffered(),
    &Port::vtpActive_410x853fec0->getBuffered(),
    &Port::vtpActive_420x853fed0->getBuffered(),
    &Port::vtpActive_430x853fee0->getBuffered(),
    &Port::vtpActive_440x853fef0->getBuffered(),
    &Port::vtpActive_450x853ff00->getBuffered(),
    &Port::vtpActive_460x853ff10->getBuffered(),
    &Port::vtpActive_470x853ff20->getBuffered(),
    &Port::vtpActive_480x853ff30->getBuffered(),
    &Port::vtpActive_490x853ff40->getBuffered()
};
hl3::PArrayWrapper<const Boolean, index_t, 11> vtpActive(_array_vtpActive);
const AnalogReal *_array_t_brake[] = {
    &Port::t_brake_650x8540040->getBuffered()
};
hl3::PArrayWrapper<const AnalogReal, index_t, 1> t_brake(_array_t_brake);
const AnalogReal *_array_x_closedTooLong_1[] = {
    &Port::x_closedTooLong_1_660x8540050->getBuffered(),
    &Port::x_closedTooLong_1_670x8540060->getBuffered(),
    &Port::x_closedTooLong_1_680x8540070->getBuffered()
};
hl3::PArrayWrapper<const AnalogReal, index_t, 3>
    x_closedTooLong_1(_array_x_closedTooLong_1);
const AnalogReal *_array_x_closedTooLong_2[] = {
    &Port::x_closedTooLong_2_690x8540080->getBuffered(),
    &Port::x_closedTooLong_2_700x853fd88->getBuffered(),
    &Port::x_closedTooLong_2_710x853fd98->getBuffered()
};
hl3::PArrayWrapper<const AnalogReal, index_t, 3>
    x_closedTooLong_2(_array_x_closedTooLong_2);

// Evaluation Interpretation
bool _result0x854bd18 = false;
{
    hl3::FixedArray<Integer, index_t, 1> i;
    for (i[0] = 0; i[0] <= (VTP_COUNT[0] - 1) && !_result0x854bd18; ++i[0]) {
        _result0x854bd18 =
            (((brakePoint[i[0]] <= x[0]) && (RA[0].vtp[i[0]].v[0] < v[0]))
             && (RA[0].vtp[i[0]].x[0] > x[0]))
            && (vtpActive[i[0]]
                || ((RA[0].vtp[i[0]].type[0] == VTP_TYPE_CROSSING)
                    && (((x_closedTooLong_1[RA[0].vtp[i[0]].crId[0]]
                        <= (RA[0].cr[RA[0].vtp[i[0]].crId[0]].x_end[0]
                            + GC[0].trainLength[0]))
                        && (t_closedTooLong[RA[0].vtp[i[0]].crId[0]]
                            <= t_brake[0]))
                    || ((x_closedTooLong_2[RA[0].vtp[i[0]].crId[0]]
                        <= (RA[0].cr[RA[0].vtp[i[0]].crId[0]].x_end[0]
                            + GC[0].trainLength[0]))
                ))
    }
}

```

```

        && (t_closedTooLong[RA[0].vtp[i[0]].crId[0]]
          > t_brake[0]))));
    }
}
return _result0x854bd18;
}

```

### D.2.14 InvariantConstraint::expression43

```

bool InvariantConstraint::expression43 () {
    ////////////////////////////////////////////////////////////////////
    // ! condAdjustCrossingSafe //
    // ////////////////////////////////////////////////////////////////////
    // ! exists c in [0..CROSSING_COUNT-1] : { //
    //   crossingSafe[c] && ((x_closedTooLong_1[c] //
    //     <= RA.cr[c].x_end + GC.trainLength //
    //     && t_closedTooLong[c] <= t_brake) //
    //   || (x_closedTooLong_2[c] //
    //     <= RA.cr[c].x_end + GC.trainLength //
    //     && t_closedTooLong[c] > t_brake)) //
    // } //
    ////////////////////////////////////////////////////////////////////

    // Initialization of Local Variables from Channels
    const RouteAtlas *_array_RA [] = {
        &Port::RA_310x853fd18->getBuffered()
    };
    hl3::PArrayWrapper<const RouteAtlas, index_t, 1> RA(_array_RA);
    const Integer *_array_CROSSING_COUNT [] = {
        &Port::CROSSING_COUNT_280x853fce8->getBuffered()
    };
    hl3::PArrayWrapper<const Integer, index_t, 1>
    CROSSING_COUNT(_array_CROSSING_COUNT);
    const GlobalConstants *_array_GC [] = {
        &Port::GC_300x853fd08->getBuffered()
    };
    hl3::PArrayWrapper<const GlobalConstants, index_t, 1> GC(_array_GC);
    const AnalogReal *_array_t_closedTooLong [] = {
        &Port::t_closedTooLong_330x853fd38->getBuffered(),
        &Port::t_closedTooLong_340x853fd48->getBuffered(),
        &Port::t_closedTooLong_350x853fd58->getBuffered()
    };
    hl3::PArrayWrapper<const AnalogReal, index_t, 3>
    t_closedTooLong(_array_t_closedTooLong);
    const Boolean *_array_crossingSafe [] = {
        &Port::crossingSafe_500x853ff50->getBuffered(),
        &Port::crossingSafe_510x853ff60->getBuffered(),
        &Port::crossingSafe_520x853ff70->getBuffered()
    };
    hl3::PArrayWrapper<const Boolean, index_t, 3>
    crossingSafe(_array_crossingSafe);
    const AnalogReal *_array_t_brake [] = {
        &Port::t_brake_650x8540040->getBuffered()
    };
    hl3::PArrayWrapper<const AnalogReal, index_t, 1> t_brake(_array_t_brake);
    const AnalogReal *_array_x_closedTooLong_1 [] = {
        &Port::x_closedTooLong_1_660x8540050->getBuffered(),
        &Port::x_closedTooLong_1_670x8540060->getBuffered(),
        &Port::x_closedTooLong_1_680x8540070->getBuffered()
    };
    hl3::PArrayWrapper<const AnalogReal, index_t, 3>
    x_closedTooLong_1(_array_x_closedTooLong_1);
}

```

```

const AnalogReal *_array_x_closedTooLong_2[] = {
    &Port::x_closedTooLong_2_690x8540080->getBuffered(),
    &Port::x_closedTooLong_2_700x853fd88->getBuffered(),
    &Port::x_closedTooLong_2_710x853fd98->getBuffered()
};
hl3::PArrayWrapper<const AnalogReal,index_t,3>
x_closedTooLong_2(_array_x_closedTooLong_2);

// Evaluation Interpretation
bool _result0x854a410 = false;
{
    hl3::FixedArray<Integer,index_t,1> c;
    for (c[0] = 0;
        (c[0] <= (CROSSING_COUNT[0] - 1)) && !_result0x854a410; ++c[0]) {
        _result0x854a410 =
            (crossingSafe[c[0]]
             && (((x_closedTooLong_1[c[0]] <= (RA[0].cr[c[0]].x_end[0]
                + GC[0].trainLength[0]))
                 && (t_closedTooLong[c[0]] <= t_brake[0]))
             || ((x_closedTooLong_2[c[0]] <= (RA[0].cr[c[0]].x_end[0]
                + GC[0].trainLength[0]))
                 && (t_closedTooLong[c[0]] > t_brake[0]))));
    }
}
return (! _result0x854a410);
}

```

### D.2.15 Guard::expression44

```

bool Guard::expression44 () {
    // true
    return true;
}

```

### D.2.16 Guard::expression45

```

bool Guard::expression45 () {
    // true
    return true;
}

```

### D.2.17 Guard::expression46

```

bool Guard::expression46 () {
    ////////////////////////////////////////////////////
    // condBrakingRequired //
    // //
    // exists i in [0..VTP_COUNT-1] : { ( //
    //   brakePoint[i] <= x && RA.vtp[i].v < v && RA.vtp[i].x > x //
    //   && (vtpActive[i] || //
    //     RA.vtp[i].type == VTP_TYPE::CROSSING //
    //     && ((x_closedTooLong_1[RA.vtp[i].crId] //
    //         <= RA.cr[RA.vtp[i].crId].x_end + GC.trainLength //
    //         && t_closedTooLong[RA.vtp[i].crId] <= t_brake) //
    //     || (x_closedTooLong_2[RA.vtp[i].crId] //
    //         <= RA.cr[RA.vtp[i].crId].x_end + GC.trainLength //
    //         && t_closedTooLong[RA.vtp[i].crId] > t_brake))) //
    //   )} //
    ////////////////////////////////////////////////////

    // Initialization of Local Variables from Channels
    const RouteAtlas *_array_RA[] = {

```

```

    &Port::RA_310x853fd18->getBuffered()
};
hl3::PArrayWrapper<const RouteAtlas, index_t, 1> RA(_array_RA);
const Integer *_array_VTP_COUNT[] = {
    &Port::VTP_COUNT_290x853fcf8->getBuffered()
};
hl3::PArrayWrapper<const Integer, index_t, 1> VTP_COUNT(_array_VTP_COUNT);
const GlobalConstants *_array_GC[] = {
    &Port::GC_300x853fd08->getBuffered()
};
hl3::PArrayWrapper<const GlobalConstants, index_t, 1> GC(_array_GC);
const AnalogReal *_array_brakePoint[] = {
    &Port::brakePoint_530x853ff80->getBuffered(),
    &Port::brakePoint_540x853ff90->getBuffered(),
    &Port::brakePoint_550x853ffa0->getBuffered(),
    &Port::brakePoint_560x853ffb0->getBuffered(),
    &Port::brakePoint_570x853ffc0->getBuffered(),
    &Port::brakePoint_580x853ffd0->getBuffered(),
    &Port::brakePoint_590x853ffe0->getBuffered(),
    &Port::brakePoint_600x853fff0->getBuffered(),
    &Port::brakePoint_610x8540000->getBuffered(),
    &Port::brakePoint_620x8540010->getBuffered(),
    &Port::brakePoint_630x8540020->getBuffered()
};
hl3::PArrayWrapper<const AnalogReal, index_t, 11>
    brakePoint(_array_brakePoint);
const AnalogReal *_array_x[] = {
    &Port::x_370x853fd78->getBuffered()
};
hl3::PArrayWrapper<const AnalogReal, index_t, 1> x(_array_x);
const AnalogReal *_array_v[] = {
    &Port::v_380x853fe90->getBuffered()
};
hl3::PArrayWrapper<const AnalogReal, index_t, 1> v(_array_v);
const AnalogReal *_array_t_closedTooLong[] = {
    &Port::t_closedTooLong_330x853fd38->getBuffered(),
    &Port::t_closedTooLong_340x853fd48->getBuffered(),
    &Port::t_closedTooLong_350x853fd58->getBuffered()
};
hl3::PArrayWrapper<const AnalogReal, index_t, 3>
    t_closedTooLong(_array_t_closedTooLong);
const Boolean *_array_vtpActive[] = {
    &Port::vtpActive_390x853fea0->getBuffered(),
    &Port::vtpActive_400x853feb0->getBuffered(),
    &Port::vtpActive_410x853fec0->getBuffered(),
    &Port::vtpActive_420x853fed0->getBuffered(),
    &Port::vtpActive_430x853fee0->getBuffered(),
    &Port::vtpActive_440x853fef0->getBuffered(),
    &Port::vtpActive_450x853ff00->getBuffered(),
    &Port::vtpActive_460x853ff10->getBuffered(),
    &Port::vtpActive_470x853ff20->getBuffered(),
    &Port::vtpActive_480x853ff30->getBuffered(),
    &Port::vtpActive_490x853ff40->getBuffered()
};
hl3::PArrayWrapper<const Boolean, index_t, 11> vtpActive(_array_vtpActive);
const AnalogReal *_array_t_brake[] = {
    &Port::t_brake_650x8540040->getBuffered()
};
hl3::PArrayWrapper<const AnalogReal, index_t, 1> t_brake(_array_t_brake);
const AnalogReal *_array_x_closedTooLong_1[] = {
    &Port::x_closedTooLong_1_660x8540050->getBuffered(),
    &Port::x_closedTooLong_1_670x8540060->getBuffered(),
};

```

```

    &Port::x_closedTooLong_1_680x8540070->getBuffered()
};
hl3::PArrayWrapper<const AnalogReal ,index_t,3>
x_closedTooLong_1(_array_x_closedTooLong_1);
const AnalogReal *_array_x_closedTooLong_2[] = {
    &Port::x_closedTooLong_2_690x8540080->getBuffered(),
    &Port::x_closedTooLong_2_700x853fd88->getBuffered(),
    &Port::x_closedTooLong_2_710x853fd98->getBuffered()
};
hl3::PArrayWrapper<const AnalogReal ,index_t,3>
x_closedTooLong_2(_array_x_closedTooLong_2);

// Evaluation Interpretation
bool _result0x854a9c8 = false;
{
    hl3::FixedArray<Integer ,index_t,1> i;
    for (i[0] = 0; (i[0] <= (VTP_COUNT[0] - 1)) && !_result0x854a9c8; ++i[0]) {
        _result0x854a9c8 =
            (((brakePoint[i[0]] <= x[0]) && (RA[0].vtp[i[0]].v[0] < v[0]))
            && (RA[0].vtp[i[0]].x[0] > x[0]))
            && (vtpActive[i[0]] ||
            ((RA[0].vtp[i[0]].type[0] == VTP_TYPE_CROSSING)
            && (((x_closedTooLong_1[RA[0].vtp[i[0]].crId[0]]
            <= RA[0].cr[RA[0].vtp[i[0]].crId[0]].x_end[0]
            + GC[0].trainLength[0]))
            && (t_closedTooLong[RA[0].vtp[i[0]].crId[0]] <= t_brake[0]))
            || ((x_closedTooLong_2[RA[0].vtp[i[0]].crId[0]]
            <= RA[0].cr[RA[0].vtp[i[0]].crId[0]].x_end[0]
            + GC[0].trainLength[0]))
            && (t_closedTooLong[RA[0].vtp[i[0]].crId[0]]
            > t_brake[0])))))));
    }
}
return _result0x854a9c8;
}

```

### D.2.18 Guard::expression48

```

bool Guard::expression48 () {
    ////////////////////////////////////////////////////////////////////
    // condAdjustCrossingSafe //
    ////////////////////////////////////////////////////////////////////
    // exists c in [0..CROSSING_COUNT-1] : { //
    //   crossingSafe[c] && ((x_closedTooLong_1[c] //
    //     <= RA.cr[c].x_end + GC.trainLength //
    //     && t_closedTooLong[c] <= t_brake) //
    //   || (x_closedTooLong_2[c] //
    //     <= RA.cr[c].x_end + GC.trainLength //
    //     && t_closedTooLong[c] > t_brake)) //
    // } //
    ////////////////////////////////////////////////////////////////////

    // Initialization of Local Variables from Channels
    const RouteAtlas *_array_RA[] = {
        &Port::RA_310x853fd18->getBuffered()
    };
    hl3::PArrayWrapper<const RouteAtlas ,index_t,1> RA(_array_RA);
    const Integer *_array_CROSSING_COUNT[] = {
        &Port::CROSSING_COUNT_280x853fce8->getBuffered()
    };
    hl3::PArrayWrapper<const Integer ,index_t,1>
    CROSSING_COUNT(_array_CROSSING_COUNT);
}

```

```

const GlobalConstants *_array_GC[] = {
    &Port::GC_300x853fd08->getBuffered()
};
hl3::PArrayWrapper<const GlobalConstants, index_t, 1> GC(_array_GC);
const AnalogReal *_array_t_closedTooLong[] = {
    &Port::t_closedTooLong_330x853fd38->getBuffered(),
    &Port::t_closedTooLong_340x853fd48->getBuffered(),
    &Port::t_closedTooLong_350x853fd58->getBuffered()
};
hl3::PArrayWrapper<const AnalogReal, index_t, 3>
    t_closedTooLong(_array_t_closedTooLong);
const Boolean *_array_crossingSafe[] = {
    &Port::crossingSafe_500x853ff50->getBuffered(),
    &Port::crossingSafe_510x853ff60->getBuffered(),
    &Port::crossingSafe_520x853ff70->getBuffered()
};
hl3::PArrayWrapper<const Boolean, index_t, 3>
    crossingSafe(_array_crossingSafe);
const AnalogReal *_array_t_brake[] = {
    &Port::t_brake_650x8540040->getBuffered()
};
hl3::PArrayWrapper<const AnalogReal, index_t, 1> t_brake(_array_t_brake);
const AnalogReal *_array_x_closedTooLong_1[] = {
    &Port::x_closedTooLong_1_660x8540050->getBuffered(),
    &Port::x_closedTooLong_1_670x8540060->getBuffered(),
    &Port::x_closedTooLong_1_680x8540070->getBuffered()
};
hl3::PArrayWrapper<const AnalogReal, index_t, 3>
    x_closedTooLong_1(_array_x_closedTooLong_1);
const AnalogReal *_array_x_closedTooLong_2[] = {
    &Port::x_closedTooLong_2_690x8540080->getBuffered(),
    &Port::x_closedTooLong_2_700x853fd88->getBuffered(),
    &Port::x_closedTooLong_2_710x853fd98->getBuffered()
};
hl3::PArrayWrapper<const AnalogReal, index_t, 3>
    x_closedTooLong_2(_array_x_closedTooLong_2);

// Evaluation Interpretation
bool _result0x8556080 = false;
{
    hl3::FixedArray<Integer, index_t, 1> c;
    for (c[0] = 0;
         c[0] <= (CROSSING_COUNT[0] - 1)) && !_result0x8556080; ++c[0]) {
        _result0x8556080 =
            (crossingSafe[c[0]]
             && (((x_closedTooLong_1[c[0]] <= (RA[0].cr[c[0]].x_end[0]
                                             + GC[0].trainLength[0]))
                 && (t_closedTooLong[c[0]] <= t_brake[0]))
             || ((x_closedTooLong_2[c[0]] <= (RA[0].cr[c[0]].x_end[0]
                                             + GC[0].trainLength[0]))
                 && (t_closedTooLong[c[0]] > t_brake[0]))));
    }
}
return _result0x8556080;
}

```

### D.2.19 Guard::expression51

```

bool Guard::expression51 () {
    ////////////////////////////////////////
    // condBrakingNotRequired //
    // //
}

```

```

// forall i in [0..VTP_COUNT-1] : { !(
//   brakePoint[i] <= x && RA.vtp[i].v < v && RA.vtp[i].x > x
//   && (vtpActive[i] ||
//     RA.vtp[i].type == VTP_TYPE::CROSSING
//     && ((x_closedTooLong_1[RA.vtp[i].crlId]
//       <= RA.cr[RA.vtp[i].crlId].x_end + GC.trainLength
//       && t_closedTooLong[RA.vtp[i].crlId] <= t_brake)
//     || (x_closedTooLong_2[RA.vtp[i].crlId]
//       <= RA.cr[RA.vtp[i].crlId].x_end + GC.trainLength
//       && t_closedTooLong[RA.vtp[i].crlId] > t_brake)))
//   )}
// }
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

// Initialization of Local Variables from Channels
const RouteAtlas *_array_RA [] = {
  &Port::RA_310x853fd18->getBuffered()
};
hl3::PArrayWrapper<const RouteAtlas, index_t, 1> RA(_array_RA);
const Integer *_array_VTP_COUNT [] = {
  &Port::VTP_COUNT_290x853fcf8->getBuffered()
};
hl3::PArrayWrapper<const Integer, index_t, 1> VTP_COUNT(_array_VTP_COUNT);
const GlobalConstants *_array_GC [] = {
  &Port::GC_300x853fd08->getBuffered()
};
hl3::PArrayWrapper<const GlobalConstants, index_t, 1> GC(_array_GC);
const AnalogReal *_array_brakePoint [] = {
  &Port::brakePoint_530x853ff80->getBuffered(),
  &Port::brakePoint_540x853ff90->getBuffered(),
  &Port::brakePoint_550x853ffa0->getBuffered(),
  &Port::brakePoint_560x853ffb0->getBuffered(),
  &Port::brakePoint_570x853ffc0->getBuffered(),
  &Port::brakePoint_580x853ffd0->getBuffered(),
  &Port::brakePoint_590x853ffe0->getBuffered(),
  &Port::brakePoint_600x853fff0->getBuffered(),
  &Port::brakePoint_610x8540000->getBuffered(),
  &Port::brakePoint_620x8540010->getBuffered(),
  &Port::brakePoint_630x8540020->getBuffered()
};
hl3::PArrayWrapper<const AnalogReal, index_t, 11>
  brakePoint(_array_brakePoint);
const AnalogReal *_array_x [] = {
  &Port::x_370x853fd78->getBuffered()
};
hl3::PArrayWrapper<const AnalogReal, index_t, 1> x(_array_x);
const AnalogReal *_array_v [] = {
  &Port::v_380x853fe90->getBuffered()
};
hl3::PArrayWrapper<const AnalogReal, index_t, 1> v(_array_v);
const AnalogReal *_array_t_closedTooLong [] = {
  &Port::t_closedTooLong_330x853fd38->getBuffered(),
  &Port::t_closedTooLong_340x853fd48->getBuffered(),
  &Port::t_closedTooLong_350x853fd58->getBuffered()
};
hl3::PArrayWrapper<const AnalogReal, index_t, 3>
  t_closedTooLong(_array_t_closedTooLong);
const Boolean *_array_vtpActive [] = {
  &Port::vtpActive_390x853fea0->getBuffered(),
  &Port::vtpActive_400x853feb0->getBuffered(),
  &Port::vtpActive_410x853fec0->getBuffered(),
  &Port::vtpActive_420x853fed0->getBuffered(),
  &Port::vtpActive_430x853fee0->getBuffered(),

```

```

    &Port::vtpActive_440x853fef0->getBuffered(),
    &Port::vtpActive_450x853ff00->getBuffered(),
    &Port::vtpActive_460x853ff10->getBuffered(),
    &Port::vtpActive_470x853ff20->getBuffered(),
    &Port::vtpActive_480x853ff30->getBuffered(),
    &Port::vtpActive_490x853ff40->getBuffered()
};
hl3::PArrayWrapper<const Boolean, index_t, 11> vtpActive(_array_vtpActive);
const AnalogReal *_array_t_brake[] = {
    &Port::t_brake_650x8540040->getBuffered()
};
hl3::PArrayWrapper<const AnalogReal, index_t, 1> t_brake(_array_t_brake);
const AnalogReal *_array_x_closedTooLong_1[] = {
    &Port::x_closedTooLong_1_660x8540050->getBuffered(),
    &Port::x_closedTooLong_1_670x8540060->getBuffered(),
    &Port::x_closedTooLong_1_680x8540070->getBuffered()
};
hl3::PArrayWrapper<const AnalogReal, index_t, 3>
x_closedTooLong_1(_array_x_closedTooLong_1);
const AnalogReal *_array_x_closedTooLong_2[] = {
    &Port::x_closedTooLong_2_690x8540080->getBuffered(),
    &Port::x_closedTooLong_2_700x853fd88->getBuffered(),
    &Port::x_closedTooLong_2_710x853fd98->getBuffered()
};
hl3::PArrayWrapper<const AnalogReal, index_t, 3>
x_closedTooLong_2(_array_x_closedTooLong_2);

// Evaluation Interpretation
bool _result0x854d580 = true;
{
    hl3::FixedArray<Integer, index_t, 1> i;
    for (i[0] = 0; (i[0] <= (VTP_COUNT[0] - 1)) && _result0x854d580; ++i[0]) {
        _result0x854d580 =
            (! (((brakePoint[i[0]] <= x[0]) && (RA[0].vtp[i[0]].v[0] < v[0]))
                && (RA[0].vtp[i[0]].x[0] > x[0]))
                && (vtpActive[i[0]]
                    || ((RA[0].vtp[i[0]].type[0] == VTP_TYPE_CROSSING)
                        && (((x_closedTooLong_1[RA[0].vtp[i[0]].crId[0]]
                            <= (RA[0].cr[RA[0].vtp[i[0]].crId[0]].x_end[0]
                                + GC[0].trainLength[0]))
                            && (t_closedTooLong[RA[0].vtp[i[0]].crId[0]]
                                <= t_brake[0]))
                    || ((x_closedTooLong_2[RA[0].vtp[i[0]].crId[0]]
                            <= (RA[0].cr[RA[0].vtp[i[0]].crId[0]].x_end[0]
                                + GC[0].trainLength[0]))
                            && (t_closedTooLong[RA[0].vtp[i[0]].crId[0]]
                                > t_brake[0]))))))));
    }
}
return _result0x854d580;
}

```

# Bibliography

- [ACH<sup>+</sup>95] Rajeev Alur, Costas Courcoubetis, Nicolas Halbwachs, Thomas A. Henzinger, Pei-Hsin Ho, Xavier Nicollin, Alfredo Olivero, Joseph Sifakis, and Sergio Yovine. The algorithmic analysis of hybrid systems. *Theor. Comput. Sci.*, 138(1):3–34, 1995.
- [ACHH93] Rajeev Alur, Costas Courcoubetis, Thomas A. Henzinger, and Pei-Hsin Ho. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In Grossman et al. [GNRR93], pages 209–229.
- [AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.
- [ADE<sup>+</sup>01] R. Alur, T. Dang, J. Esposito, R. Fierro, Y. Hur, F. Ivančić, V. Kumar, I. Lee, P. Mishra, G. Pappas, and O. Sokolsky. Hierarchical hybrid modeling of embedded systems. *Lecture Notes in Computer Science*, 2211:14–31, 2001.
- [ADE<sup>+</sup>03] R. Alur, T. Dang, J. Esposito, Y. Hur, F. Ivančić, V. Kumar, I. Lee, P. Mishra, G. Pappas, and O. Sokolsky. Hierarchical hybrid modeling and analysis of embedded systems. *Proceedings of the IEEE*, 91(1):11–28, January 2003.
- [AGLS01] Rajeev Alur, Radu Grosu, Insup Lee, and Oleg Sokolsky. Compositional refinement for hierarchical hybrid systems. In *Proceedings of the 4th International Workshop on Hybrid Systems: Computation and Control*, volume 2034 of *Lecture Notes in Computer Science*, pages 33–48, 2001.
- [Amt99] Peter Amthor. *Structural Decomposition of Hybrid Systems. Test Automation for Hybrid Reactive Systems*. PhD thesis, Universität Bremen, BISS Monographs, No. 13, October 1999.
- [And96] C. André. Representation and analysis of reactive behaviors: A synchronous approach. In *Computational Engineering in Systems Applications (CESA)*, pages 19–29, Lille (F), July 1996. IEEE-SMC.
- [AO97] Krzysztof R. Apt and Ernst-Rüdiger Olderog. *Verification of sequential and concurrent programs (2nd ed.)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1997.

- [BBB<sup>+</sup>99] Tom Bienmüller, Jürgen Bohn, Henning Brinkmann, Udo Brockmeyer, Werner Damm, Hardi Hungar, and Peter Jansen. Verification of automotive control units. In *Correct System Design*, volume 1710 of *Lecture Notes in Computer Science*, pages 319–341, 1999.
- [BBHP03] Kirsten Berkenkötter, Stefan Bisanz, Ulrich Hannemann, and Jan Peleska. HybridUML Profile for UML 2.0. SVERTS Workshop at the «UML» 2003 Conference, October 2003. <http://www-verimag.imag.fr/EVENTS/2003/SVERTS/>.
- [BBHP04a] Kirsten Berkenkötter, Stefan Bisanz, Ulrich Hannemann, and Jan Peleska. Executable HybridUML and its Application to Train Control Systems. In Ehrig et al. [EDD<sup>+</sup>04], pages 145–173.
- [BBHP04b] Kirsten Berkenkötter, Stefan Bisanz, Ulrich Hannemann, and Jan Peleska. Spezifikation von Echtzeit-Automatisierungssystemen mit HybridUML. *atp – Automatisierungstechnische Praxis*, 46(8):54–60, August 2004. ISSN 0178-2320.
- [Ber00] G. Berry. *The Foundations of Esterel*. MIT Press, 2000.
- [Ber03] Kirsten Berkenkötter. Using UML 2.0 in real-time development - a critical review. SVERTS Workshop at the «UML» 2003 Conference, October 2003. <http://www-verimag.imag.fr/EVENTS/2003/SVERTS/>.
- [BF99] Stefan Bisanz and Ingo Fiß. Grafischer Entwurf von CSP-Spezifikationen für den Test eingebetteter Echtzeitsysteme. Master's thesis, Universität Bremen, February 1999.
- [BG92] Gerard Berry and Georges Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.
- [Bin00] Robert V. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison-Wesley, 2000.
- [BT02a] Stefan Bisanz and Aliko Tsiolakis. Test Development in Virtual Environments. In Dominik Haneberg, Gerhard Schellhorn, and Wolfgang Reif, editors, *FM-TOOLS 2002. The 5th Workshop on Tools for System Design and Verification*, pages 65–69, Augsburg, June 2002. Universität Augsburg, Institut für Informatik. Technical Report 2002-11.
- [BT02b] Stefan Bisanz and Aliko Tsiolakis. Using a Virtual Reality Environment to Generate Test Specifications. In Rob Hierons and Thierry Jéron, editors, *Formal Approaches to Testing of Software. FATES'02 – A Satellite Workshop of CONCUR'02*, pages 121–135. INRIA, August 2002.
- [BZL04] Stefan Bisanz, Paul Ziemann, and Arne Lindow. Integrated Specification, Validation and Verification with HybridUML and OCL applied to the BART Case Study. In Eckehard Schnieder and Géza

- Tarnai, editors, *FORMS/FORMAT 2004. Formal Methods for Automation and Safety in Railway and Automotive Systems*, pages 191–203, Braunschweig, December 2004. Proceedings of Symposium FORMS/FORMAT 2004, Braunschweig, Germany, 2nd and 3rd December 2004. ISBN 3-9803363-8-7.
- [CHA] CHARON toolkit and information. <http://www.cis.upenn.edu/mobies/charon/>. Department of Computer and Information Science, University of Pennsylvania.
- [Cho78] Tsun S. Chow. Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering*, SE-4(3):178–186, March 1978.
- [CJR95] Zhou Chaochen, Wang Ji, and Anders P. Ravn. A formal description of hybrid systems. In Rajeev Alur, Thomas A. Henzinger, and Eduardo D. Sontag, editors, *Hybrid Systems*, volume 1066 of *Lecture Notes in Computer Science*, pages 511–530. Springer, 1995.
- [DFG] Priority Programme Software Specification – Integration of Software Specification Techniques for Applications in Engineering. <http://fs.cs.tu-berlin.de/projekte/indspec/SPP>.
- [Dij76] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.
- [DJHP98] Werner Damm, Bernhard Josko, Hardi Hungar, and Amir Pnueli. A compositional real-time semantics of STATEMATE designs. *Lecture Notes in Computer Science*, 1536:186–238, 1998.
- [Dou99] Bruce Powel Douglass. *Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns*. Addison Wesley Longman, Inc., May 1999. ISBN 0-201-49837-5.
- [Dou04] Bruce Powel Douglass. *Real-Time UML Third Edition. Advances in the UML for Real-Time Systems*. Addison-Wesley Professional, February 2004. ISBN 0-321-16076-2.
- [EDD<sup>+</sup>04] Hartmut Ehrig, Werner Damm, Jörg Desel, Martin Große-Rhode, Wolfgang Reif, Eckehard Schnieder, and Engelbert Westkämper, editors. *Integration of Software Specification Techniques for Applications in Engineering, Priority Program SoftSpez of the German Research Foundation (DFG), Final Report*, volume 3147 of *Lecture Notes in Computer Science*. Springer, September 2004.
- [Efk05] Christof Efke. Development and evaluation of a hard real-time scheduling modification for linux 2.6. Master’s thesis, Universität Bremen, April 2005.
- [EFS02] S. Engell, G. Frehse, and E. Schnieder, editors. *Modelling, Analysis and Design of Hybrid Systems*, volume 279 of *Lecture Notes in Control and Information Sciences*. Springer Verlag, 2002. ISBN 3-540-43812-2.

- [EMO] Hilding Elmquist, Sven Erik Mattsson, and Martin Otter. Object-oriented and hybrid modeling in modelica.
- [FFB96] Betriebliches Lastenheft für den FunkFahrBetrieb. Requirements Specification of the Radio-Based Train Control, Deutsche Bahn AG, Undisclosed Document, October 1996.
- [GGBL91] P. Le Guernic, T. Gautier, M. Le Borgne, and C. Lemaire. Programming real-time applications with signal. *Proc. of the IEEE*, 79(9):1321–1336, September 1991.
- [GNRR93] Robert L. Grossman, Anil Nerode, Anders P. Ravn, and Hans Rischel, editors. *Hybrid Systems*, volume 736 of *Lecture Notes in Computer Science*. Springer, 1993.
- [GS98] Radu Grosu and Thomas Stauner. Modular and visual specification of hybrid systems – an introduction to hycharts. Technical Report TUM-I9801, Technische Universität München, 1998.
- [Hal92] Nicolas Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer Academic Publishers, Norwell, MA, USA, 1992.
- [Har87] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.
- [HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.
- [Hen96] Thomas A. Henzinger. The theory of Hybrid Automata. In *Proceedings of the 11th Annual Symposium on Logic in Computer Science (LICS)*, pages 278–292. IEEE Computer Society Press, 1996.
- [HHK03] Thomas A. Henzinger, Benjamin Horowitz, and Christoph M. Kirsch. Giotto: A time-triggered language for embedded programming. *Proceedings of the IEEE*, 91:84–99, 2003.
- [HHWT97] Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. HYTECH: A model checker for hybrid systems. *International Journal on Software Tools for Technology Transfer*, 1(1–2):110–122, 1997. Available at <http://www-cad.eecs.berkeley.edu/~tah/HyTech/>.
- [HL94] Constance Heitmeyer and Nancy Lynch. The generalized railroad crossing: A case study in formal verification of real-time systems. In *IEEE Real-Time Systems Symposium*, pages 120–131. IEEE Computer Society, 1994.
- [HN96] David Harel and Amnon Naamad. The STATEMATE semantics of statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, October 1996.
- [Hoa85] Charles Anthony Richard Hoare. *Communicating Sequential Processes*. Prentice Hall International, Hemel Hempstead, 1985.

- [HP02] A. E. Haxthausen and J. Peleska. A Domain Specific Language for Railway Control Systems. In *Proceedings of the Sixth Biennial World Conference on Integrated Design and Process Technology, (IDPT2002), Pasadena, California, June 23-28 2002*.
- [HP03] A. E. Haxthausen and J. Peleska. Automatic Verification, Validation and Test for Railway Control Systems based on Domain-Specific Descriptions. In *Proceedings of the 10th IFAC Symposium on Control in Transportation Systems*. Elsevier Science Ltd, Oxford, 2003.
- [HPSS04] Frank Hänsel, Jan Poliak, Roman Slovák, and Eckehard Schnieder. Reference case study "traffic control systems" for comparison and validation of formal specifications using a railway model demonstrator. In Ehrig et al. [EDD<sup>+</sup>04], pages 96–118.
- [Int96] International Organization for Standardization. *ISO/IEC 14977:1996: Information technology — Syntactic metalanguage — Extended BNF*. International Organization for Standardization, Geneva, Switzerland, 1996.
- [Jif94] He Jifeng. From CSP to hybrid systems. pages 171–189, 1994.
- [Jos95] Mathai Joseph. *Real-Time Systems: Specification, Verification, and Analysis*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1995.
- [JS00] L. Jansen and E. Schnieder. Traffic control systems case study: Problem description and a note on domain-based software specification, 2000.
- [KMP00] Y. Kesten, Z. Manna, and A. Pnueli. Verification of clocked and hybrid systems. *Acta Informatica*, 36(11):836–912, 2000.
- [Kop97] Hermann Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. The Kluwer International Series in Engineering and Computer Science. Kluwer Academic Publishers, 1997.
- [KP92] Y. Kesten and A. Pnueli. Timed and hybrid statecharts and their textual representation. In J. Vytupil, editor, *Formal Techniques in Real-Time and Fault-Tolerant Systems 2nd International Symposium*, volume 571, pages 591–619, Nijmegen, The Netherlands, 1992. Springer-Verlag.
- [LSV03] N. A. Lynch, R. Segala, and F. W. Vaandrager. Hybrid I/O automata. *Information and Computation*, 185(1):105–157, August 2003.
- [MAT] MATLAB. <http://www.mathworks.com>. The MathWorks, Inc.
- [Mey01] Oliver Meyer. *Structural Decomposition of Timed CSP and its Application in Real-Time Testing*. PhD thesis, Universität Bremen, BISS Monographs, No. 16, July 2001.

- [MR01] F. Maraninchi and Y. Rémond. Argos: an automaton-based synchronous language. *Computer Languages*, (27):61–92, 2001.
- [Nor03] André Nordwig. *Integration von Sichten für die objektorientierte Modellierung hybrider Systeme*. PhD thesis, Technische Universität Berlin, Berlin, 2003. ISBN 3-89825-692-8.
- [NOSY93] Xavier Nicollin, Alfredo Olivero, Joseph Sifakis, and Sergio Yovine. An approach to the description and analysis of hybrid systems. In Grossman et al. [GNRR93], pages 149–178.
- [OH05] Aliko Ott and Tobias Hartmann. Domain Specific V&V Strategies for Aircraft Applications. In *Proc. of the 6th ICSTEST International Conference on Software Testing*, Düsseldorf, April 2005.
- [OMGa] Object Management Group. Human-Usable Textual Notation (HUTN) specification. <http://www.uml.org>.
- [OMGb] Object Management Group. Meta-Object Facility (MOF) specification. <http://www.omg.org/mof>.
- [OMGc] Object Management Group. Object Constraint Language (OCL) specification. <http://www.uml.org>.
- [OMGd] Object Management Group. UML Profile for Schedulability, Performance, and Time Specification. <http://www.uml.org>.
- [OMGe] Object Management Group. XML Metadata Interchange (XMI) specification. <http://www.uml.org>.
- [OMGO03] Object Management Group (OMG). Meta Object Facility (MOF) 2.0 Core Specification, October 2003.
- [Ope] OpenModelica. <http://www.ida.liu.se/~pelab/modelica/OpenModelica.html>. Linköping University, Sweden.
- [Ott06] Aliko Ott. *System Testing in the Avionics Domain*. PhD thesis, Universität Bremen, 2006. To appear.
- [Pac02] Joern Pachl. *Railway Operation and Control*. VTD Rail Publishing, Mountlake Terrace (USA), 2002. ISBN 0-9719915-1-0.
- [PBF99] J. Peleska, S. Bisanz, I. Fiß, and M. Endreß. Non-Standard Graphical Simulation Techniques for Test Specification Development. In H. Szczerbicka, editor, *Modelling and simulation: A tool for the next millenium. 13th European Simulation Multiconference 1999*, volume 1, pages 575–580, Delft, 1999. Society for Computer Simulation International.
- [Pel96] Jan Peleska. *Formal Methods and the Development of Dependable Systems*. Number 9612. Christian-Albrechts-Universität Kiel, Institut für Informatik und Praktische Mathematik, December 1996. Habilitationsschrift.
- [Rat] Rational Rose. <http://www.ibm.com/software/rational>. IBM.

- [Rav95] A. P. Ravn. Design of embedded real-time computing systems. Technical Report ID-TR 1995-170, ID/DTU, Lyngby, Denmark, October 1995. dr. techn. dissertation.
- [RGG94] Duke R., Rose G., and Smith G. Object-Z: A specification language advocated for the description of standards. Technical report 94-45, Software Verification Research Centre, School of Information Technology, The University of Queensland, Brisbane 4072. Australia, December 1994.
- [Rha] Rhapsody. <http://www.ilogix.com>. I-Logix Inc.
- [Ric02] Mark Richters. *A Precise Approach to Validating UML Models and OCL Constraints*. PhD thesis, Universität Bremen, Logos Verlag, Berlin, BISS Monographs, No. 14, 2002.
- [RJB04] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language – Reference Manual, 2nd edition*. Addison-Wesley, July 2004.
- [RRS03] Mauno Rönkkö, Anders P. Ravn, and Kaisa Sere. Hybrid Action Systems. *Theoretical Computer Science*, 290:937–973, January 2003.
- [RTA] Real-Time Application Interface (RTAI). <http://www.rtai.org>. Originally founded by the Department of Aerospace Engineering of Politecnico di Milano (DIAPM), <http://www.aero.polimi.it/~rtai/about/index.html>.
- [RTL] RTLinux. <http://www.rtlinux.org>. FSMLabs, Inc.
- [Sca] Scade Suite. <http://www.esterel-technologies.com>. Esterel Technologies, Inc.
- [Sel98] Bran Selic. Using UML for Modeling Complex Real-Time Systems. In *Languages, Compilers, and Tools for Embedded Systems, ACM SIGPLAN Workshop LCTES'98, Montreal, Canada, June 1998, Proceedings*, volume 1474 of *Lecture Notes in Computer Science*, pages 250–260. Springer, 1998.
- [Sim] Simulink. <http://www.mathworks.com>. The MathWorks, Inc.
- [Spi92] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science, 2nd edition, 1992.
- [Sto96] Neil Storey. *Safety-Critical Computer Systems*. Addison Wesley Longman, 1996.
- [Tog] Borland Together. <http://www.borland.com/together>. Borland Software Corporation.
- [UML] Object Management Group. *Unified Modeling Language (UML) Specification*. Available at <http://www.uml.org>.

- [vdB94] M. von der Beeck. A comparison of StateCharts variants. In *Proc. of Formal Techniques in Real Time and Fault Tolerant Systems*, pages 128–148, Berlin, 1994. Springer-Verlag.
- [VS04] Verified Systems. RT-Tester 6.x – User Manual. Technical Report Verified-INT-014-2003, Verified Systems International GmbH, Bremen, 2004.
- [VxW] VxWorks. <http://www.windriver.com>. Wind River.
- [WB01] Victor L. Winter and Sourav Bhattacharya. *High Integrity Software*. Kluwer Academic Publishers Press., 2001.
- [YLB94] Y. Y. Yang, Derek A. Linkens, and S. P. Banks. Modelling of hybrid systems based on Extended Coloured Petri Nets. In Panos J. Antsaklis, Wolf Kohn, Anil Nerode, and Shankar Sastry, editors, *Hybrid Systems*, volume 999 of *Lecture Notes in Computer Science*, pages 509–528. Springer, 1994.
- [ZRH93] Chaochen Zhou, A. P. Ravn, and M. R. Hansen. An extended duration calculus for hybrid real-time systems. In R. L. Grossman, A. Nerode, A. P. Ravn, and H. Rischel, editors, *Hybrid Systems*, volume 763 of *Lecture Notes in Computer Science*, pages 36–59. The Computer Society of the IEEE, 1993. Extended abstract.