

Architectural Refinement in HETS

Mihai Codescu

Dissertation

zur Erlangung des Grades eines Doktors der
Ingenieurwissenschaften

– Dr.-ing. –

Vorgelegt im Fachbereich 3 (Mathematik und Informatik)
der Universität Bremen

Mai 2012

Datum des Promotionskolloquiums:

16.08.2012

Gutachter:

Prof. Dr. Till Mossakowski, Bremen

Prof. Dr. Andrzej Tarlecki, Warschau

Acknowledgments

I would firstly like to thank my supervisor, Till Mossakowski, for his incredible support and encouragement and for contributing as a discussion partner to the ideas presented in this work. I am also grateful to him for developing HETS, thus allowing me to alternate between theoretical and implementation work.

Many of the results presented here are the outcome of collaboration. I would like to thank my coauthors: Till Mossakowski, Adrian Riesco, Bruno Langenstein, Christian Maeder, Daniel Găină, Florian Rabe, Fulya Horozal, Gregor Horsinka, Kristina Sojakova, Michael Kohlhase, Oliver Kutz and Rafaela Rau for giving me the chance to benefit from working with them. Dominik Lücke, Lutz Schröder and Andrzej Tarlecki provided valuable feedback.

I would like to thank all the members of AG Krieg-Brückner for the friendly environment and the good working atmosphere. Christian Maeder deserves special thanks for his constant help in deciphering the insides of HETS. I also wish to thank Florian Rabe, Fulya Horozal, Kristina Sojakova and Michael Kohlhase, for their cooperation within the LATIN project.

Special thanks go to Răzvan Diaconescu and my former colleagues at Școala Normală Superioară București: Andrei Popescu, Daniel Găină, Denisa Diaconescu and Marius Petria for introducing me to the field of algebraic specifications and for their friendship.

I am indebted to Olivier Commowick for creating and making available online the \LaTeX template used for editing this thesis.

I gratefully acknowledge the financial support of DFKI GmbH Bremen and of the Deutsche Forschungsgemeinschaft (DFG) under grant MO 971/2-1 - project LATIN.

Contents

I	Introduction and Preliminaries	9
1	Introduction	11
1.1	Algebraic Specifications	11
1.2	Heterogeneous Specifications	13
1.3	Stepwise Refinement in CASL	14
1.4	Objectives of the Thesis	16
1.5	Roadmap	18
1.6	Publications	19
2	HETS	23
2.1	Architecture of HETS	24
2.2	HETS' Logic Graph	26
3	Mathematical Foundations	33
3.1	Institutions	33
3.1.1	Colimits and Amalgamation	38
3.1.2	Institutions with Proofs	39
3.2	Institution Comorphisms	40
3.3	Grothendieck Institutions	43
4	CASL	45
4.1	CASL Logic	45
4.2	Structured Specifications	47
4.3	Development Graphs	51
4.4	CASL Architectural Specifications	53
4.5	HETCASL	58
5	CASL Refinement Language	61
5.1	Syntax	61
5.2	Semantics	68
II	HETS Development	77
6	Logical Frameworks in HETS	79
6.1	Preliminaries	80
6.1.1	Proof-Theoretic Logical Frameworks	80
6.1.2	A Logic Atlas in LF	82
6.2	The LATIN Metaframework	84
6.2.1	Logical Meta-Frameworks	85

6.2.2	Generalizations	89
6.3	Logical Frameworks in HETS	90
6.3.1	Implementing the LMF in HETS	90
6.3.2	LF as a Logical Framework in HETS	92
6.3.3	Adding a New Logic in HETS: FOL	92
6.4	Conclusion and Future Work	93
7	Generalized Theoroidal Comorphisms	95
7.1	Theoroidal Comorphisms	95
7.2	Motivating Examples	97
7.3	Generalized Comorphisms	101
7.4	Heterogeneous Specifications	102
7.5	Heterogeneous Proofs	104
7.6	Conclusions	106
8	Heterogeneous Colimits and Applications	107
8.1	Exactness in Grothendieck Institutions	107
8.2	Example: Heterogeneous Ontologies	109
8.3	Relaxing Colimits and Amalgamation	110
8.4	Algorithms for the Relaxed Setting	112
8.5	Colimits in CASL	116
8.6	Normal Forms of Specifications with Hiding	118
8.7	Conclusion	120
9	Integrating Maude into HETS	123
9.1	Rewriting Logic and Maude	124
9.2	Relating Maude and CASL Logics	126
9.2.1	Maude Logic	126
9.2.2	Encoding Maude in CASL	127
9.3	From Maude Modules to Development Graphs	128
9.4	Normalization of free definition links	130
9.5	An example: reversing lists	134
9.6	Conclusions and Future Work	136
III	Architectural Refinement in HETS	137
10	Lambda Expressions in Architectural Specifications	139
10.1	Semantics of Generic Unit Expressions	140
10.2	Adding Dependency Tracking	143
10.3	Completeness of Extended Static Semantics	148
10.4	Parametric Architectural Specifications	149
10.5	Refinement of Units with Imports	153
10.6	An Application: Warehouse System	156
10.7	Conclusions	158

11 Correctness and Consistency of CASL Refinements	159
11.1 Proof Calculus for CASL Architectural Specifications	160
11.2 Proof Calculus for CASL Refinements	171
11.3 Completeness of the Proof Calculus for Refinements	178
11.4 Refinement Trees	184
11.5 Checking Consistency of Refinement Specifications	188
11.6 Conclusion and Future Work	191
12 VSE Refinement in HETS	195
12.1 Presentation of VSE	196
12.2 Institution of Dynamic Logic	198
12.2.1 Signatures	198
12.2.2 Sentences	199
12.2.3 Models	201
12.2.4 Satisfaction of Dynamic Logic Formulas	201
12.2.5 Satisfaction of Procedure Definitions	203
12.2.6 Satisfaction of restricted sort generation constraints	203
12.2.7 Satisfaction condition	204
12.3 VSE Refinement as an Institution Comorphism	205
12.3.1 The Refinement Comorphism	207
12.3.2 Structuring in Context of Refinement	211
12.4 Example: Implementing natural numbers by binary words	213
12.5 Conclusions and future work	219
IV Final Remarks	221
13 Conclusions and Future Work	223
13.1 Summary	223
13.2 Future Work	224
Bibliography	227
V Appendices	241
A Natural numbers as binary words	243
B Steam boiler control system	247

Part I

Introduction and Preliminaries

Introduction

Contents

1.1 Algebraic Specifications	11
1.2 Heterogeneous Specifications	13
1.3 Stepwise Refinement in CASL	14
1.4 Objectives of the Thesis	16
1.5 Roadmap	18
1.6 Publications	19

In this chapter, we first present the basic concepts of algebraic specification of software systems. We see that there are many logical systems in use, to cover different aspects of the software systems. This amounts to the need for a heterogeneous framework for specification, where specifications written in different logics coexist and can be combined in a formal way. Such a framework is provided by the Heterogeneous Tool Set HETS, which is based on an extension of the de-facto standard algebraic specification language CASL. The central research topic of this work is to further extend HETS, both theoretically and implementation-wise, in two directions. First we will present experiences and new insights on its foundations from developing HETS with new features. We then concentrate on the notion of stepwise refinement, which has been a topic of research for many years in the theory of algebraic specification, but was missing as a syntactical construction in CASL and thus in HETS.

After an introductory discussion on algebraic specification, heterogeneity and refinement of specifications, we present our goals in Sec. 1.4 and introduce the roadmap of this thesis in Sec. 1.5. The chapter is then concluded with the list of articles in which parts of this work have been published.

1.1 Algebraic Specifications

Formal methods for software engineering refer to the use of mathematical techniques and tools in ensuring that a certain software product is correct. They are applied especially in the area of safety-critical systems, where it is very important that the system is indeed safe. Formal methods can be employed already from the early stages of the development process, starting with the specification level, when a description of the expected behavior of the system is produced. The advantage of

using a formal language for writing specifications is that no ambiguities are introduced, as the denotation of a specification is defined precisely, in a mathematical way. Moreover, we can mathematically ensure that a certain design or implementation are correct with respect to the formal specification that we fixed before.

The basic assumption underlying the algebraic techniques of specification is that programs are regarded as algebras, that is, a collection of sets of data values and functions over those sets. The main purpose is to concentrate on the functional behavior of software systems and to verify correctness of the input/output behavior of a program. In this approach, a specification consists of logical axioms which describe the expected properties an implementation should satisfy. Its denotation however is model-theoretical: a specification determines a class of models satisfying the axioms. The intention is to associate to each program a model in the logical system and thus a program is a correct realization of a specification if its associated model is included in the model class determined by the specification.

Originally, many-sorted equational logic was used for specifying abstract data types axiomatically. However, a large number of extensions and variations have been developed, to increase expressivity and to cover more aspects of software systems. This has greatly influenced the design of algebraic specification languages even from the very first one, CLEAR [Burstall and Goguen, 1980], as the features of the language like e.g. structuring operations for specification “in the large” can be separated from the particularities of the logical system of choice (see also [Sannella and Tarlecki, 1988b]). This has the advantage that the same specification language can be (re-)used for any choice of underlying logical system and moreover, the logic-dependent aspects can be abstracted away, allowing thus a better understanding of the general theory underlying algebraic specification. Towards this purpose, Goguen and Burstall introduced the concept of *institution* [Goguen and Burstall, 1992], a category theoretic formulation of the notion of logical system which allows to develop both the theory of algebraic specifications and the features of algebraic specification languages at an abstract, *institution-independent* level.

Starting with the late seventies, many algebraic specification languages have been developed. We refer the interested reader to [Wirsing, 1995] and to Chapter 8 of [Astesiano et al., 2000] for an overview. The Common Algebraic Specification Language CASL [Mosses, 2004] has been designed by the “Common Framework Initiative for Algebraic Specification and Development” with the goal to provide a standard language for algebraic specification, expressive enough to subsume many of the existing languages. CASL provides

- a logic for writing individual specifications, namely multi-sorted partial first-order logic with subsorting and induction;
- structuring constructs for specifications and a formalism for structured theorem proving, both institution-independent;
- as a novel feature, *architectural specifications* for prescribing the structure of

the implementation of a system, again in an institution-independent way, as we will see in more detail below;

- libraries of specifications to ease re-use of specifications and their distribution on the Web.

1.2 Heterogeneous Specifications

As also mentioned above, a great number of logical systems are used in formal specification. The reason is that, on one side, some logics are better suited for certain types of problems and on the other side, software systems have reached a such degree of complexity that in some cases it may be useful to give different specifications in different formalisms for the same component of a software system, to cover different aspects of the application (“viewpoint specification”). Moreover, there are research communities focused on each of the formalisms and investing efforts in providing specialized tools for the underlying logical system. The philosophy of heterogeneous specification is not to attempt to combine all features of all different logics into a single “universal” logic, but rather to allow them to coexist and to provide a framework for translating between formalisms during the specification and verification processes. The specifier has thus the freedom to choose the logic that suits best the problem to be solved, offers best tool support and according to the degree of familiarity with a certain specification language. A very simple example is the trade-off between the higher expressivity of higher-order logic when compared to first-order logic and the better automated proof support for the latter logic: when writing the specification, the user would prefer to use first-order logic whenever possible and thus benefit from an automated mechanism for discharging the proof obligations introduced in the verification process, and higher-order logic only whenever necessary. Another goal of a heterogeneous framework is to be easily extensible with new formalisms.

A number of approaches to heterogeneous specification have been proposed, CafeOBJ [Futatsugi and Diaconescu, 1998] providing the example of a heterogeneous algebraic specification language. Heterogeneity is achieved in CafeOBJ by representing several logical systems, among which the logics of order-sorted algebras, hidden algebras, preorder algebras and their combinations, as institutions and the translations between these logics as so-called institution morphisms [Goguen and Roşu, 2002], thus obtaining a cube of logics and projections between them, similarly to the known lambda-cube in type theory. This cube is then flattened by applying the Grothendieck construction [Diaconescu, 2002] to it and thus obtaining again an institution.

The Heterogeneous Tool Set HETS [Mossakowski, 2005] is a natural extension of CASL to heterogeneous specification. As we have seen above, the structuring, proof management, architectural and library levels of CASL are independent of the underlying institution and the Grothendieck construction provides a way of flattening a graph of logics and their translations (this time not only projections be-

tween logics can be considered, but various kinds of other translations as well, see [Mossakowski, 2002a, 2003, 2005, 2006]) to obtain an institution. This institution is then used at the basic level of CASL to obtain heterogeneous specification in HETS, and the underlying specification language is called HETCASL. Note that the graph of logics in HETS is much more complex than the one of CafeOBJ and the specialized tools of the formalisms included in this graph are interfaced by HETS to ensure proof support. [Wölfl et al., 2007] presents an example that fits precisely the scenario above, namely a heterogeneous verification using HETS of the RCC8 composition table – specified in first-order logic – w.r.t. the interpretation in metric spaces – specified in higher-order logic. This verification task can be split into two subtasks: (1) the interactive verification (using Isabelle) of just a few bridge lemmas, and (2) the automatic verification that the bridge lemmas imply the RCC8 composition table (using SPASS). It should be stressed that HETS eases the integration of new logics, and that its logics are not limited to algebraic specification techniques.

The work presented in this thesis has a very deep connection with HETS. The entire picture presented so far lies at a high level of abstraction and HETS provides a concrete and very dynamic instance of framework for heterogeneous specification. The goals of this thesis are on one side, to extend the theory underlying HETS with new results and on the other side, to present experiences learned from extending the implementation of HETS with features previously developed, as it was often the case that this led to new insights and reconsideration on theory from a practical perspective.

1.3 Stepwise Refinement in CASL

The task of verifying correctness of a program w.r.t. the requirements given in a specification may be hard to do directly in practice, because the program usually reflects a number of design decisions like choice of data structures, of a certain algorithm and so on. A cleaner methodology is to start with an abstract requirement specification, that fixes only the expected properties and leaves open any such design decision, and to incorporate these decisions into the specification in a stepwise manner while proving correctness of each step. Each choice narrows the model class of the specification being refined. The process ends when we reach a specification that contains enough detail to be translated into a program, and correctness of the entire development is ensured by compositionality of correct realizations. This can be represented graphically as follows:

$$SP_0 \rightsquigarrow SP_1 \rightsquigarrow \dots \rightsquigarrow SP_n \rightsquigarrow P$$

where SP_1, \dots, SP_n are specifications, P is a program implementing SP_n and we denote the refinement steps with \rightsquigarrow . This approach has been largely advocated in the literature on algebraic specification, including [Ehrig et al., 1982, Goguen and Meseguer, 1982, Sannella and Tarlecki, 1988a]; in software engineering the

idea can be traced back until at least [Wirth, 1971]. We refer the reader to Chapter 7 of [Astesiano et al., 2000] and to [Sannella and Tarlecki, 1997] for a survey. Note that it may be not straightforward to translate from the last specification to a program, as this may e.g. depend on the target programming language, see [Sannella and Tarlecki, 1996] for a discussion on this topic. This task is however outside the scope of algebraic specification in general, and of CASL in particular. Actually, the linear model is too simple to cover the practical situations. An important aspect concerns modular decomposition of a complex system into several implementation tasks, that can be resolved independently, e.g. by different developers. This corresponds to introducing branching points in the refinement tree. CASL architectural specifications [Bidoit et al., 2002a, Mosses, 2004] have been designed for this purpose, based on the insight that the structure of implementations may be different from the structure of specifications [Fitzgerald and Jones, 1990].

$$SP \rightsquigarrow BR \left\{ \begin{array}{l} SP_1 \\ \vdots \\ SP_n \end{array} \right.$$

Each branching point additionally specifies an operation BR that combines realizations P_1, \dots, P_n of SP_1, \dots, SP_n respectively to produce a realization of SP . As explained in [Bidoit et al., 2002a], the operation BR can be thought of as a linking procedure, in the sense that whenever we want to replace an implementation of a component with another one, we can simply "re-link" it with the realizations of the other components, without modifying them in any way. This is possible because the development of components is independent.

CASL architectural specifications allow though the specification of individual branching points only, and we may want to further decompose the specifications of the components as well. In particular, this should be possible without having to rewrite already defined architectural specifications. Moreover, CASL does not provide any syntactical construct for specifying explicitly refinement between specifications. [Mossakowski et al., 2005] introduce an extension of CASL with a simple refinement language that adds the means to formalize whole developments in the form of *refinement trees*.

$$SP \rightsquigarrow \left\{ \begin{array}{l} SP_1 \rightsquigarrow P_1 \\ \vdots \\ SP_n \rightsquigarrow \left\{ \begin{array}{l} SP_{n1} \rightsquigarrow \{ SP_{n11} \rightsquigarrow P_{n11} \\ \dots \\ SP_{nm} \rightsquigarrow P_{nm} \end{array} \end{array} \right.$$

Furthermore, the refinement notion has been enhanced to a behavioral or observational interpretation. This has been again widely studied in literature, starting with [Giarratana et al., 1976], see also [Sannella and Tarlecki, 1987] and [Sannella

and Tarlecki, 1997] for an overview. The idea is that an implementation is considered correct even if it does not satisfy the properties of a specification literally, but only up to observable behavior. This is achieved e.g. in many-sorted logics by treating only a subset of sorts in the specification as directly observable while the others are regarded as hidden and can be “observed” by terms of an observable sort and by predicates. It is then possible to introduce an observational congruence relation on the submodel obtained by restricting the carriers of hidden sorts to the values of closed terms, which is then used as interpretation of equality. The classical example is that in the implementation of stacks as arrays with pointers, two arrays are regarded to be equal if they only differ in their “junk” entries, above the stack pointer. This has been developed in theory to some degree [Bidoit et al., 2008, Sannella and Tarlecki, 2012], but not implemented in HETS yet.

1.4 Objectives of the Thesis

The aims of this thesis are twofold:

- to instantiate and further develop a number of theoretical results in the concrete setting provided by HETS as a framework for heterogeneous specification and
- to add support for the CASL refinement language in HETS, including a first approach to observational refinement.

For the first objective, our investigations target the following directions.

Logical frameworks in HETS. HETS uses institutions to formalize the notion of a logical system. This allows, on one side, to develop foundational results and even language constructs independently of the underlying logical system and, on the other side, to implement institutions in HETS as a generic interface which is then instantiated whenever a new logical system is added. Thus, the effort of adding new logics is considerably reduced. However, this has the drawback that new logics can be added only by HETS developers. Moreover, logics are not represented as formal objects in HETS, and their verification can not be done automatically, but only informally at the meta-level. Logical frameworks like LF or Isabelle on the other hand provide strong capabilities for representing proof theory of a logic. In his PhD thesis [Rabe, 2008], F. Rabe has developed a meta-framework for representing logics which is no longer biased towards proof theory, but includes a representation of the model theory of an object logic as well. Our goal is to integrate this meta-framework in HETS, making it possible to add new logics in HETS in a more declarative manner, from their specification in a logical framework.

Logic translations. The next central notion in HETS is the one of translation between logics. Translations are represented in HETS as institution comorphisms,

which usually capture embeddings or encodings of logics. In their theoroidal variant, comorphisms map theories of the source logic to theories of the target logic, under the conditions that theories of the same signature are mapped to theories of the same signature and the translation of sentences depends on signatures, not on theories [Meseguer, 1989]. However, the logic graph of HETS includes several logic translations that do not have these properties. We propose a generalization of the notion of theoroidal comorphisms and study the impact of this change to heterogeneous specification.

Heterogeneous colimits and amalgamation. Colimits are a categorical notion that is used to combine interconnected objects taking into account the interconnection. They have been used as a means for putting together logical theories and specifications (see e.g. [Burstall and Goguen, 1977, Diaconescu et al., 1993]). A major property of colimits of specifications is *amalgamation*. Roughly speaking, this property states that models of given specifications can be combined to yield a uniquely determined model of a colimit specification, provided that the original models coincide on common components. While studied before [Diaconescu, 2002], the conditions for existence of colimits and amalgamation in a heterogeneous setting prove to be too strong for all the practical situations. We develop an algorithm for computing approximations of amalgamable colimits based on a result in [Mossakowski, 2005] and also discuss their applications in HETS.

Maude integration. Maude [Clavel et al., 2007] is a high-performance system, supporting both equational and rewriting logic for a wide range of applications. Moreover, it provides another instance of logical framework for representing logics (see [Martí-Oliet and Meseguer, 1994]). We present here an integration of Maude as a new logic in HETS. For HETS, this brings the first dedicated rewriting engine; for Maude, this enables external proof support at a general level using the provers interfaced by HETS. We also include an extension of the proof management system of HETS to cope with the special semantics of modules in Maude.

The second objective has also developed in more directions.

Semantics of generic unit expressions. In architectural specifications, generic units denote parameterized programs, which combine compatible implementations taken as arguments to obtain an implementation of an extension of their specifications, in a way that the arguments are preserved. They are build using generic unit expressions, written in CASL using the λ -notation. We identify a shortcoming in the analysis of the semantics of generic unit expressions, rather technical, but which has also an impact on the way units with imports can be refined. We propose a number of changes to remedy this.

CASL refinement in HETS. The refinement language of CASL subsumes the architectural level and allows to capture the whole development process as a refinement

tree. We implement the language in HETS and complement it with a calculus for verifying correctness of refinement trees, which are also represented explicitly in the tool. We also investigate conditions for completeness of the calculus. Moreover, we present a calculus for checking consistency of refinements. In particular, this calculus provides a method of simplifying the task of finding a model for a large theory using an architectural refinement of this theory.

Integration of VSE. The specification environment Verification Support Environment (VSE) [Autexier et al., 2000], developed at DFKI Saarbrücken, provides an industrial-strength methodology for specification and verification of imperative programs. VSE supports a notion of refinement of first-order specifications to specifications in dynamic logic, where modalities are programs written in an imperative language. VSE further exploits this to provide code generation, making thus refinement to VSE a good candidate for the last step of the development. This refinement notion mimics behavioral refinement in a somewhat simpler way, based on making submodels and congruences explicit. Our goal here is to integrate VSE and its refinement notion into HETS in a non-disruptive way w.r.t. the proof management system of HETS.

1.5 Roadmap

The thesis is organized as follows. The rest of Part I is dedicated to setting the foundations. We start in Chapter 2 with a presentation of the Heterogeneous Tool Set HETS and of the formalisms it currently supports. Chapter 3 recalls the mathematical concepts underlying HETS, starting with institutions and translations between them like institutions, model amalgamation or conservativity of theory morphisms, institution comorphisms and Grothendieck institutions. In Chapter 4 we present the CASL language, starting with the underlying logic and then the structuring and the architectural levels, together with the formalism of development graphs for proof management. Chapter 5 describes the CASL refinement language and its semantics.

In Part II we concentrate on the first objective, the extensions of HETS, both from a theoretical and from the implementation perspective. In Chapter 6 we describe how Hets can be extended with new logics from their specification in a logical framework. Chapter 7 introduces a new notion of generalized theoroidal comorphisms and discusses the impact of this change to heterogeneous specification and verification. In Chapter 8 we present the implementation of an algorithm for computing approximations of heterogeneous amalgamable colimits and its applications in HETS. Finally, Chapter 9 describes the integration of the Maude logic in HETS and extensions of the development graph calculus introduced to facilitate proof support for Maude. Note that Part II depends on Chapters 2, 3 and 4.

Part III focuses on the architectural refinement level of CASL, our second objective. Chapter 10 discusses a modification in the semantics of the architectural specifications which simplifies verification and refinement of unit with imports. In

Chapter 11 we introduce a proof calculus for verification of CASL architectural and refinement specifications, discuss its completeness and complement it with a calculus for checking consistency of refinements. Chapter 12 presents the integration of the VSE refinement notion in HETS. Note that chapters 4 and 5 are a prerequisite for understanding Part III, but the other chapters of Part I are also relevant.

Fig. 1.1 gives a preview of the contributions of this work and their dependencies. The diagram is organized according to the levels of HETCASL, starting with basic specifications (BS in the diagram), then with structured specifications and development graphs (DG), architectural specifications (AS), refinement specifications (RS) and finally heterogeneous specifications (HS). In each box, the number points to the corresponding chapter or section. Note that heterogeneous refinement and its incoming edges are dashed because future work is required to support this entirely – however the part needed for writing down VSE refinements using the refinement language of CASL is already available.

1.6 Publications

Parts of this thesis have been published in the following articles:

1. Mihai Codescu, Till Mossakowski (2008). **Heterogeneous colimits**. In F. Boulanger, C. Gaston, P. Schobbens (Eds.), MoVaH'08 Workshop on Modeling, Validation and Heterogeneity. IEEE press.
2. Mihai Codescu (2009). **Generalized Theoroidal Institution Comorphisms**. In Andrea Corradini, Ugo Montanari (Eds.), WADT 2008, Vol. 5486, pp. 88-101, Lecture Notes in Computer Science. Springer.
3. Mihai Codescu, Bruno Langenstein, Christian Maeder, Till Mossakowski (2009). **The VSE Refinement Method in HETS**. In K. Breitman, A. Cavalcanti (Eds.), ICFEM 2009, Vol. 5885, pp. 660-678, Lecture Notes in Computer Science. Springer. *Superseded by 8*.
4. Mihai Codescu, Till Mossakowski, Adrian Riesco, Christian Maeder (2010). **Integrating Maude into HETS**. In Mike Johnson, Dusko Pavlovic (Eds.), AMAST 2010, Vol. 6486, pp. 60-75, Lecture Notes in Computer Science. Springer.
5. Mihai Codescu (2011). **Lambda Expressions in CASL Architectural Specifications**. In Till Mossakowski, Hans-Jörg Kreowski (Eds.), Recent Trends in Algebraic Development Techniques, 20th International Workshop, WADT 2010, Lecture Notes in Computer Science. Springer.
6. Mihai Codescu, Fulya Horozal, Michael Kohlhase, Till Mossakowski, Florian Rabe, Kristina Sojakova.(2011) **Towards Logical Frameworks in the Heterogeneous Tool Set HETS**. In Till Mossakowski, Hans-Jörg Kreowski

(Eds.), *Recent Trends in Algebraic Development Techniques*, 20th International Workshop, WADT 2010, Lecture Notes in Computer Science. Springer.

7. Mihai Codescu, Till Mossakowski (2011). **Refinement trees: calculi, tools and applications**. In Andrea Corradini, Bartek Klin (Eds.), *Algebra and Coalgebra in Computer Science, CALCO'11*, Vol. 6859, pp. 145-160, Lecture Notes in Computer Science. Springer.
8. Mihai Codescu, Bruno Langenstein, Christian Maeder, Till Mossakowski (2012). **The VSE Refinement Method in HETS**. To appear in *Electr. Comm. of the EASST*.

The relationship between these papers and the present thesis, as well as my own contribution to each of them is clarified at the end of the corresponding chapters. Moreover, the following papers have been published during my PhD studies but the results have not been included in the present thesis:

1. Mihai Codescu, Daniel Găină (2008). **Birkhoff Completeness in Institutions**. In *Logica Universalis*, 2(2), pp. 277-309.
2. Oliver Kutz, Till Mossakowski, Mihai Codescu (2008). **Shapes of Alignments - Construction, Combination, and Computation**. In Ulrike Sattler, Andrei Tamilin (Eds.), *International Workshop on Ontologies: Reasoning and Modularity (WORM-08)*, Vol. 348, CEUR-WS online proceedings.
3. Mihai Codescu, Fulya Horozal, Michael Kohlhase, Till Mossakowski, Florian Rabe (2011). **A Proof Theoretic Interpretation of Model Theoretic Hiding**. In *Recent Trends in Algebraic Development Techniques*, 20th International Workshop, WADT 2010, Lecture Notes in Computer Science. Springer.
4. Mihai Codescu, Fulya Horozal, Michael Kohlhase, Till Mossakowski, Florian Rabe (2011). **Project Abstract: Logic Atlas and Integrator (LATIN)**. In James H. Davenport, William M. Farmer, Josef Urban, Florian Rabe (Eds.), *Intelligent Computer Mathematics 18th Symposium, Calculemus 2011, and 10th International Conference, MKM 2011*, Bertinoro, Italy, July 18-23, 2011. Proceedings, Vol. 6824, pp. 289-291, Lecture Notes in Computer Science. Springer-Verlag Berlin Heidelberg.
5. Mihai Codescu, Gregor Horsinka, Oliver Kutz, Till Mossakowski, Rafaela Rau (2011). **DO-ROAM: Activity-Oriented Search and Navigation with OpenStreetMap**. In C. Claramunt, S. Levashkin, M. Bertolotto (Eds.), *Fourth International Conference on GeoSpatial Semantics*, Vol. 6631, (p. 88-107). , Lecture Notes in Computer Science. Springer
6. Till Mossakowski, Mihai Codescu, Oliver Kutz (2011). **Ontologie-basierte Routenplanung für eine aktivitätsorientierte Elektromobilität mit OpenStreetMap**. In *Magdeburger Logistiktagung 2011*.

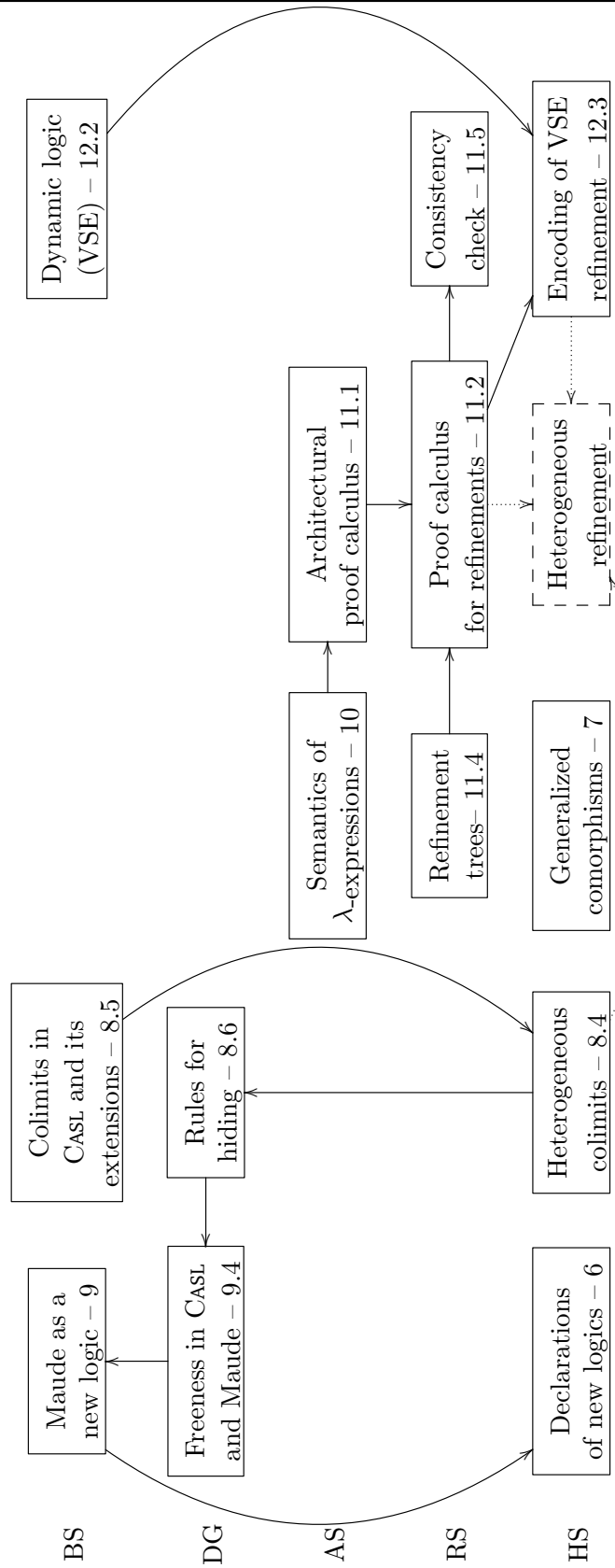


Figure 1.1: Graph of dependencies between objectives.

Contents

2.1 Architecture of HETS	24
2.2 HETS' Logic Graph	26

The Heterogeneous Tool Set HETS is a parsing, static analysis and proof management tool combining various such tools for individual specification languages, thus providing a tool for heterogeneous multi-logic specification. HETS is a both flexible, multi-lateral *and* formal (i.e. based on a mathematical semantics) integration tool. Unlike other tools, it treats logic translations as first-class citizens. The central idea of HETS is to provide a general multi-formalism framework for algebraic specification, verification and proof management. Probably the best intuition is given by the analogy of HETS acting like a motherboard where different expansion cards can be plugged in, the expansion cards here being individual logics (with their analysis and proof tools) as well as logic translations while the slots are generic interfaces.

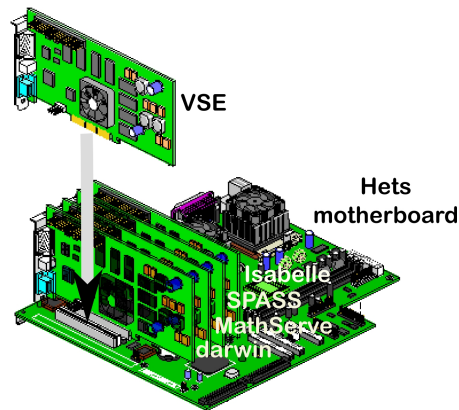


Figure 2.1: HETS motherboard.

HETS has developed into a complex tool. Therefore, we have chosen to separate its presentation in three parts. The tool itself and the formalisms it currently supports are presented in this chapter, in Sections 2.1 and 2.2 respectively. Note that this chapter follows [Mossakowski et al., 2011] to a large extent. The logical foundations of HETS are presented in detail in [Mossakowski, 2005]. We give an overview in Chapter 3, concentrating on the concepts used in the present work.

Finally, in Chapter 4 we recall the central logic of HETS, CASL, together with the CASL specification language and its extension to heterogeneous specifications, HET-CASL. As the refinement language of CASL is a rather recent addition and plays an important role in this thesis, we present it separately in Chapter 5.

2.1 Architecture of HETS

The architecture of the Heterogeneous Tool Set is shown in Fig. 2.2, which separates the logic-specific level, on the left, the logic-independent level, on the right, and the logic graph of HETS, placed in the middle and acting like a parameter for the entire tool. HETS uses Haskell [Peyton-Jones, 2003] as implementation language; we present here only some relevant implementation details and refer the interested reader to the HETS development page¹.

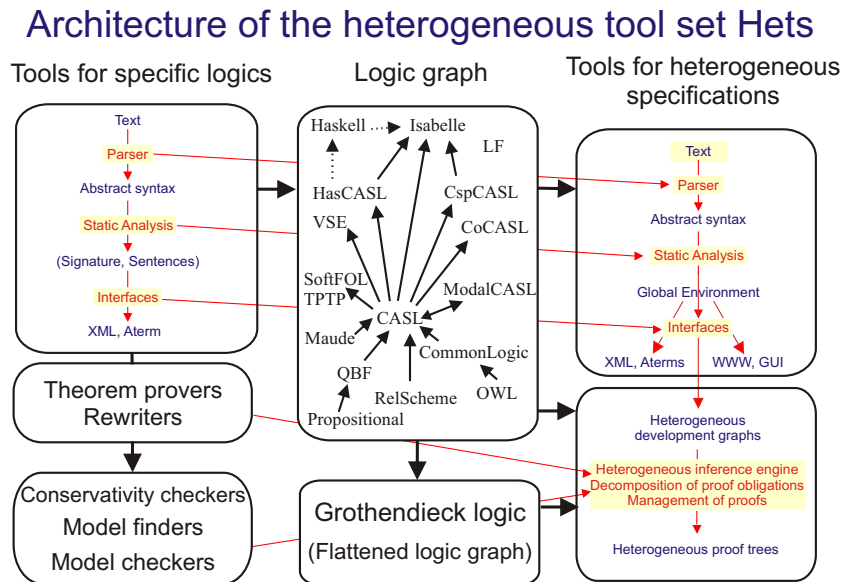


Figure 2.2: Architecture of the Heterogeneous Tool Set

HETS is based on theory of *institutions* (which we discuss in detail in Section 3.1), a model-theoretic abstract concept that formalizes logical systems. Informally, an institution provides notions of signatures for the symbols of the language, signature morphisms to capture changes of notations for symbols, then, for each signature, sentences, models and a satisfaction relation between these which is required to be preserved under change of notation.

The left side of Fig. 2.2 summarizes how an individual logic is represented internally in HETS. In our motherboard analogy, this is the expansion card for adding new logics. First, we need to provide Haskell datatypes for the constituents of

¹ <http://trac.informatik.uni-bremen.de:8080/hets/wiki/HetsForDevelopers>


```

class Logic lid sign morphism sen basic_spec symbol_map
  | lid -> sign morphism sen basic_spec symbol_map where
  identity :: lid -> sign -> morphism
  compose :: lid -> morphism -> morphism -> morphism
  dom, codom :: lid -> morphism -> sign
  parse_basic_spec :: lid -> String -> basic_spec
  parse_symbol_map :: lid -> String -> symbol_map
  parse_sentence   :: lid -> String -> sen
  basic_analysis  :: lid -> sign -> basic_spec -> (sign, [sen])
  stat_symbol_map :: lid -> sign -> symbol_map -> morphism
  map_sentence    :: lid -> morphism -> sen -> sen
  provers        :: lid -> [(sign, [sen]) -> [sen] -> Proof_status]
  cons_checkers  :: lid -> [(sign, [sen]) -> Proof_status]

```

Figure 2.3: The basic ingredients of logics

the logic, e.g. signatures, morphisms and sentences. This is done via instantiating various Haskell type classes, namely `Category` (for the signature category of the institution), `Sentences` (for the sentences), `Syntax` (for abstract syntax of basic specifications, and a parser transforming input text into this abstract syntax), `StaticAnalysis` (for the static analysis, turning basic specifications into theories, where a theory is a signature and a set of sentences). All this is assembled in the type class `Logic`, which additionally provides logic-specific tools like provers and consistency checkers. The type class `Logic` used to represent logics in Hets internally is presented in Fig. 2.3, in a simplified version. Note that with this mechanism, a new logic can only be added on the developer side; in Chapter 6 we will present an extension of HETS which allows the logics to be specified by the user, in a more declarative fashion.

The logic-independent level has a similar architecture, but the specification language is now an extension of CASL to heterogeneous specification, called HETCASL. HETCASL generalizes the CASL structuring mechanisms to arbitrary logics and allows specifications written in different logics to be combined in a formal manner. The idea is that the heterogeneous parser carries in its state the current logic, which can be altered explicitly or by translation to another logic. This determines which logic-specific parser and static analysis method will be invoked. As a result of the static analysis, HETS constructs heterogeneous *development graphs*, which provide a formalism for proof management and theorem proving. We will present HETCASL and development graphs in detail in Chapter 4. Moreover, HETS also accepts directly a number of input languages with their own structuring language, among which OWL, Maude, Haskell or Twelf. In some cases, this is realized by calling a logic-specific tool which does the parsing and the static analysis, and the result of the analysis is then imported in HETS and transformed in a development graph. This has the advantage that a number of complex algorithms do not have to be reimplemented and maintained on the HETS side.

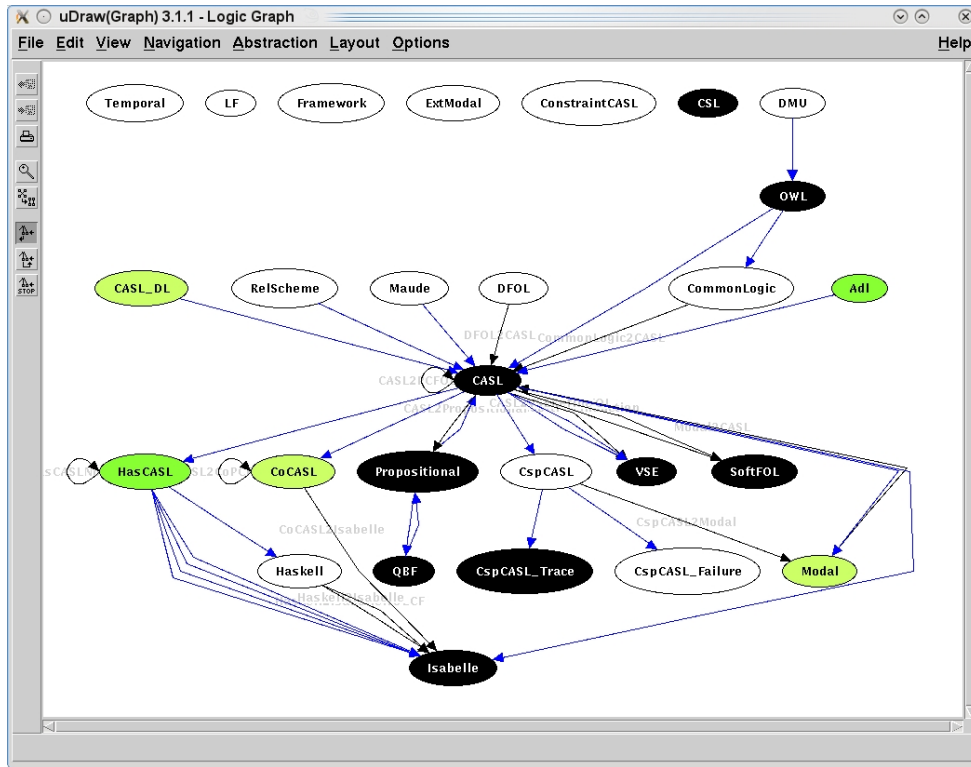


Figure 2.4: Graph of logics currently supported by HETS.

2.2 HETS' Logic Graph

In the center of HETS' architecture (Fig. 2.2) we have the graph of supported logics and logic translations. It acts like a parameter for the entire tool, as modifications on the graph do not require reimplementing of the logic-independent part of the tool. The graph of currently supported logics is in Fig. 2.4. Black nodes represent logics with proof-support, while green nodes represent stable implementations.

The logics in the logic graph of HETS can be categorized as follows:

- **general-purpose logics:** Propositional, QBF, SoftFOL, CASL, HasCASL, HOL-Light, FPL
- **logical frameworks:** Isabelle, LF, DFOL, Framework
- **ontologies and constraint languages:** CASL-DL, OWL2, CommonLogic, RelScheme, ConstraintCASL
- **logics of reactive systems:** CspCASL, CoCASL, ModalCASL, ExtModal, Maude
- **programming languages:** Haskell, VSE

- **logics of specific tools:** Reduce, DMU (CATIA), Adl, EnCL, FreeCAD

We now briefly describe each of them and also provide a reference when available.

CASL [Mosses, 2004, Bidoit and Mosses, 2004] is the central logic of HETS. We will describe this logic in more detail in Chapter 4.

CoCASL [Mossakowski et al., 2006b] is a coalgebraic extension of CASL, suited for the specification of process types and reactive systems. The central proof method is coinduction.

ModalCASL [Mossakowski, 2004] is an extension of CASL with multi-modalities and term modalities. It allows the specification of modal systems with Kripke's possible worlds semantics. It is also possible to express certain forms of dynamic logic.

ExtModal [Gârlea, 2011] is an extended modal logic, currently in an experimental state.

HasCASL is a higher order extension of CASL allowing polymorphic datatypes and functions. It is closely related to the programming language Haskell and allows program constructs being embedded in the specification. An overview of HASCASL is given in [Schröder and Mossakowski, 2002]; the language is summarized in [Schröder et al., 2003], the semantics in [Schröder, 2006, Schröder, 2005].

Haskell is a modern, pure and strongly typed functional programming language. Since it also is the implementation language of HETS, in the future HETS might be applied to itself. The definitive reference for Haskell is [Peyton-Jones, 2003], see also www.haskell.org.

CspCASL [Roggenbach, 2006] is a combination of CASL with the process algebra CSP.

Common Logic (http://en.wikipedia.org/wiki/Common_logic) is a family of languages based on first-order logic, defined as an ISO standard for knowledge representation in computer-based systems.

ConstraintCASL is an experimental logic for the specification of qualitative constraint calculi.

OWL2 is the Web Ontology Language recommended by the World Wide Web Consortium (W3C, <http://www.w3c.org>); see [w3c, 2009]. It is used for knowledge representation on the Semantic Web [Berners-Lee et al., 2001]. Hets calls an external OWL2 parser written in Java to obtain the abstract syntax for an OWL file and its imports. The Java parser also does a first analysis classifying the OWL ontology into the sublanguages OWL Full (all of OWL, under

the RDF semantics, undecidable [Schneider, 2009]), OWL DL (all of OWL, under the direct semantics [Motik et al., 2009b]), and the so-called OWL Profiles (i.e. proper sublanguages) OWL EL, OWL QL, and OWL RL [Motik et al., 2009a]. Hets supports all except OWL Full.

CASL-DL [Lüttich et al., 2005] is an extension of a restriction of CASL, realizing a strongly typed variant of OWL in CASL syntax. It extends CASL with cardinality restrictions for the description of sorts and unary predicates. The restrictions are based on the equivalence between CASL-DL, OWL and *SHOIN*. Compared to CASL only unary and binary predicates, predefined datatypes and concepts (subsorts of the topsort Thing) are allowed. It is used to bring OWL and CASL closer together.

Propositional is classical propositional logic, with the zChaff SAT solver [Herbststritt, 2003] connected to it.

QBF are quantified boolean formulas, with DepQBF (<http://fmv.jku.at/depqbf/>) connected to it.

RelScheme is a logic for relational databases [Schorlemmer and Kalfoglou, 2008].

SoftFOL [Lüttich and Mossakowski, 2007] offers several automated theorem proving (ATP) systems for first-order logic with equality:

- i) SPASS [Weidenbach et al., 2002], see <http://www.spass-prover.org>;
- ii) Vampire [Riazanov and Voronkov, 2002] see <http://www.vprover.org>;
- iii) Eprover [Schulz, 2002], see <http://www.e prover.org>;
- iv) E-KRHyper [Pelzer and Wernhard, 2007], see <http://www.uni-koblenz.de/~bpelzer/ekrhyper>, and
- v) MathServe Broker² [Zimmer and Autexier, 2006].

These together comprise some of the most advanced theorem provers for first-order logic. SoftFOL is essentially the first-order interchange language TPTP [Sutcliffe, 2010], see <http://www.tptp.org>.

Isabelle [Nipkow et al., 2002] is an interactive theorem prover for higher-order logic.

HolLight (<http://www.cl.cam.ac.uk/~jrh13/hol-light/>) is John Harrison's interactive theorem prover for higher-order logic.

VSE is an interactive theorem prover for a variant of dynamic logic, see Chapter 12.

DMU is a dummy logic to read output of "Computer Aided Three-dimensional Interactive Application" (Catia).

²which chooses an appropriate ATP upon a classification of the FOL problem

FreeCAD is a logic to read design files of the CAD system FreeCAD
<http://sourceforge.net/projects/free-cad>.

Maude (<http://maude.cs.uiuc.edu/>) is a rewrite system for first-order logic, see Chapter 9.

DFOL is an extension of first-order logic with dependent types [Rabe, 2006].

LF is the dependent type theory of Twelf (<http://twelf.plparty.org/>).

Framework is a dummy logic added for declarative logic definitions, see Chapter 6.

Adl is “A Description Language” based on relational algebra originally designed for requirements engineering of business rules (https://lab.cs.ru.nl/BusinessRules/Requirements_engineering).

Fpl is the “logic for functional programs” defined in [Sannella and Tarlecki, 2012].

EnCL is an “engineering calculation language” based on first order theory of real numbers with some predefined binders [Dietrich et al., 2011]. It allows the formulation of executable specifications of engineering calculation methods. For the execution of these specifications Hets provides connections to the computer algebra systems Mathematica, Maple and Reduce.

Various logics are supported with proof tools. Proof support for the other logics can be obtained by using logic translations to a prover-supported logic.

Logic translations translate from a given source logic to a given target logic. More precisely, one and the same logic translation may have several source and target *sublogics*: for each source sublogic, the corresponding sublogic of the target logic is indicated.

A number of translations between the logics of HETS are *logic inclusions*: CASL2CoCASL, CASL2CoCASL, CASL2HasCASL, CASL2Isabelle (inclusion of sublogic $CFOL^=$, translation (7) of [Mossakowski, 2002b]), CASL2Modal, CASL2VSE (inclusion of sublogic $CFOL$), CASL_DL2CASL, CASL2CspCASL, CspCASL2CspCASL_Failure, CspCASL2CspCASL_Trace, OWL2CASL, OWL2CommonLogic, Propositional2CASL, Propositional2QBF, QBF2Propositional, RelScheme2CASL.

Another type of translations are *encoding of features* of a logic to a “poorer” sublogic of itself: CASL2PCFOL (coding of subsorting (SubPCFOL=) by injections, see Chap. III:3.1 of [Mosses, 2004]), CASL2Propositional (translation of propositional FOL), CASL2SubCFOL (coding of partial functions by error elements, translation (4a') of [Mossakowski, 2002b], but extended to subsorting, i.e. sublogic SubPCFOL=), DFOL2CASL (translating dependent types), CoCASL2CoPCFOL (coding of subsorting by injections, similar to CASL2PCFOL), CoCASL2CoSubCFOL (coding of partial functions by error supersorts, similar to CASL2SubCFOL), HasCASL2HasCASLNoSubtypes (coding out subtypes), HasCASL2HasCASLPrograms

(coding of HASCASL axiomatic recursive definitions as HASCASL recursive program definitions).

The other logic translations currently supported by HETS are:

- Adl2CASL (inclusion taking relations to CASL predicates)
- CASL2PCFOLTopSort (coding of subsorting (SulPeCFOL=) by a top sort and unary predicates for the subsorts)
- CASL2SoftFOL (coding of CASL.SuleCFOL=E to SoftFOL [Lüttich and Mossakowski, 2007], mapping types to soft types)
- CASL2SoftFOLInduction (same as CASL2SoftFOL but with instances of induction axioms for all proof goals)
- CASL2SoftFOLInduction2 (similar to CASL2SoftFOLInduction but replaces goals with induction premises)
- CASL2VSEImport (inclusion on sublogic CFOL=)
- CASL2VSERefine (refining translation of CASL.CFOL= to VSE)
- CoCASL2Isabelle (extended translation similar to CASL2Isabelle)
- CommonLogic2CASL (Coding Common Logic to CASL. Module elimination is applied before translating to CASL.)
- CommonLogic2CASLCompact (Coding compact Common Logic to CASL. Compact Common Logic is a sublogic of Common Logic where no sequence markers occur. Module elimination is applied before translating to CASL.)
- CommonLogicModuleElimination (Eliminating modules from a Common Logic theory resulting in an equivalent specification without modules.)
- CspCASL2Modal (keeps the CASL part and interprets the CspCASL labeled transition system semantics as a Kripke structure)
- DMU2OWL (interpreting Catia output as OWL)
- HasCASL2Haskell (translation of HASCASL recursive program definitions to Haskell)
- HasCASL2IsabelleOption (coding of HasCASL to Isabelle/HOL [Gröning, 2005])
- Haskell2Isabelle (coding of Haskell to Isabelle/HOL [Torrini et al., 2007])
- Haskell2IsabelleHOLCF (coding of Haskell to Isabelle/HOLCF [Torrini et al., 2007])
- HolLight2Isabelle (coding of HolLight to Isabelle/HOL)

- Maude2CASL (encoding of rewrites as predicates)
- Modal2CASL (the standard translation of modal logic to first-order logic [Blackburn et al., 2001])

Mathematical Foundations

Contents

3.1 Institutions	33
3.1.1 Colimits and Amalgamation	38
3.1.2 Institutions with Proofs	39
3.2 Institution Comorphisms	40
3.3 Grothendieck Institutions	43

This chapter is dedicated to presenting the logical foundations underlying HETS. We assume that the reader is familiar with basic set theory and category theory. For categories, we largely follow the notations in [Mac Lane, 1971], with the notable exception that we prefer diagrammatic order for composition and denote it “;”. We denote \mathbf{Set} the category of sets and functions and \mathcal{CAT} the category of categories and functors.¹ The opposite of a category C is denoted C^{op} . If C is a category, we denote $|C|$ the class of its objects and $C(a, b)$ the class of arrows in C of source a and target b , where a, b are objects of C .

3.1 Institutions

The central abstraction principle used by HETS is to formalize logical systems as *institutions* [Goguen and Burstall, 1992], a model-theoretic notion that arose in the late 1970ies when Goguen and Burstall developed a semantics for the modular specification language CLEAR [Burstall and Goguen, 1980].

We recall informally this notion here. An institution provides

- a notion of signature, carrying the context of user-defined (i.e. non-logical) symbols, and a notion of signature morphisms (translations between signatures), capturing changes of notation;
- for each signature, notions of sentence and model, and a satisfaction relation between these;
- for each signature morphism, a sentence translation and a model reduction (the direction of the latter being opposite to the signature morphism), such

¹Strictly speaking, \mathcal{CAT} is not a category but only a so-called quasi-category, which is a category that lives in a higher set-theoretic universe [Adámek et al., 1990].

that satisfaction is invariant under translation resp. reduction along signature morphisms. This has been summarised as *Truth is invariant under change of notation on context and enlargement of context*.

This leads to the following formal definition.

Definition 3.1.1 An institution $I = (\mathbf{Sig}^I, \mathbf{Sen}^I, \mathbf{Mod}^I, \models^I)$ consists of

- a category \mathbf{Sig}^I of signatures,
- a functor $\mathbf{Sen}^I : \mathbf{Sig}^I \rightarrow \mathbf{Set}$ giving, for each signature Σ , the set of sentences $\mathbf{Sen}^I(\Sigma)$, and for each signature morphism $\sigma : \Sigma \rightarrow \Sigma'$, the sentence translation map $\mathbf{Sen}^I(\sigma) : \mathbf{Sen}^I(\Sigma) \rightarrow \mathbf{Sen}^I(\Sigma')$, where often $\mathbf{Sen}^I(\sigma)(e)$ is written as $\sigma(e)$,
- a functor $\mathbf{Mod}^I : (\mathbf{Sig}^I)^{op} \rightarrow \mathbf{CAT}$ giving, for each signature Σ , the category of models $\mathbf{Mod}^I(\Sigma)$, and for each signature morphism $\sigma : \Sigma \rightarrow \Sigma'$, the reduct functor $\mathbf{Mod}^I(\sigma) : \mathbf{Mod}^I(\Sigma') \rightarrow \mathbf{Mod}^I(\Sigma)$, where often $\mathbf{Mod}^I(\sigma)(M')$ is written as $M'|_\sigma$,
- a satisfaction relation $\models_\Sigma^I \subseteq |\mathbf{Mod}^I(\Sigma)| \times \mathbf{Sen}^I(\Sigma)$ for each $\Sigma \in \mathbf{Sig}^I$,

such that for each $\sigma : \Sigma \rightarrow \Sigma'$ in \mathbf{Sig}^I the following satisfaction condition holds:

$$M' \models_{\Sigma'}^I \sigma(e) \Leftrightarrow M'|_\sigma \models_\Sigma^I e$$

for each $M' \in \mathbf{Mod}^I(\Sigma')$ and $e \in \mathbf{Sen}^I(\Sigma)$. □

We will omit the index I when it is clear from the context. Note that we make the general assumption that two isomorphic models satisfy the same sentences.

In the following we will present several examples of institutions, ranging from standard examples like propositional logic and unsorted first-order logic, to more exotic ones like an institution of relational schemes for relational databases or an institution for an imperative programming language.

Example 3.1.2 (Propositional logic) The signatures of propositional logic \mathbf{Prop} are sets Σ of propositional symbols, and signature morphisms are just maps $\sigma : \Sigma_1 \rightarrow \Sigma_2$ between such sets. A Σ -model is a function $M : \Sigma \rightarrow \{\text{True}, \text{False}\}$, and the reduct of a Σ_2 -model M_2 along a signature morphism $\sigma : \Sigma_1 \rightarrow \Sigma_2$ is the Σ_1 -model given by the composition $\sigma; M_2$. Σ -sentences are built from the propositional symbols with the usual connectives, and sentence translation is replacing the propositional symbols along the morphism. Finally, the satisfaction relation is defined by the standard truth-tables semantics. It is straightforward to see that the satisfaction condition holds.

Example 3.1.3 (First-order logic) [Goguen and Burstall, 1992] In the institution \mathbf{FOL}_{ms} of many-sorted first-order logic with equality, signatures are many-sorted first-order signatures, consisting of sorts and typed function and predicate symbols. Signature morphisms map symbols such that typing is preserved. Models are many-sorted first-order structures. Sentences are first-order formulas. Sentence translation means replacement of the translated symbols. Model reduct means reassembling the model's components according to the signature morphism. Satisfaction is the usual satisfaction of a first-order sentence in a first-order structure.

Example 3.1.4 (Partial first-order logic) [Mossakowski, 2002b] and [Burmeister, 1982]. Signatures consist of a set of sorts and sets of total and partial operations and predicates symbols, divided by their profile. Signature morphisms map the sorts and the symbols in a compatible way, and such that the totality of operation symbols is preserved. Models are first-order structures, interpreting sorts as sets, operation symbols as total/partial functions and predicates as relations. First-order sentences are built from the atomic ones, using the usual first-order logic features (connectives and quantification). Atomic sentences are predications, existential and strong equations and definedness assertions. The satisfaction of formulas is the Tarskian first-order satisfaction. One can check that thus we defined an institution, denoted \mathbf{PFOL}_{ms} .

Example 3.1.5 (Higher-order logic.) [Borzyszkowski, 1999] The many-sorted variant of higher-order logic with equality, denoted \mathbf{HOL}_{ms} , extends \mathbf{FOL}_{ms} with higher-order types, which are interpreted as appropriate subsets of the function types, where appropriate means that all λ -terms can be interpreted (Henkin semantics). Sentences extend first-order sentences by λ -abstraction and arbitrary application.

Example 3.1.6 (Relational Schemes) [Schorlemmer and Kalfoglou, 2008] Relational Schemes (for relational databases). A signature of \mathbf{Rel} consists of a set of relation symbols, where each relation symbol is indexed with a string of field names. Signature morphisms map relation symbols and field names. A model consists of a domain (set), and an n -ary relation for each relation symbol with n fields. A model reduction just forgets the parts of a model that are not needed. A sentence is a relationship between two field names of two relation symbols. Sentence translation is just renaming. A relationship is satisfied in a model if for each element occurring in the source field component of a tuple in the source relation, the same element also occurs in the target field component of a tuple in the target relation.

Example 3.1.7 (Description logic \mathcal{ALC}) [Baader et al., 2003] Signatures of the description logic \mathcal{ALC} consist of a set of \mathcal{A} of atomic concepts, a set \mathcal{R} of roles and a set \mathcal{I} of individuals, while signature morphisms σ provide respective mappings $\sigma^{\mathcal{A}}$, $\sigma^{\mathcal{R}}$ and $\sigma^{\mathcal{I}}$. Sentences are:

- TBox sentences, which are subsumption relations $C_1 \sqsubseteq C_2$ between concepts, where concepts follow the grammar

$$C ::= \mathcal{A} \mid \top \mid \perp \mid C_1 \sqcup C_2 \mid C_1 \sqcap C_2 \mid \neg C \mid \forall R.C \mid \exists R.C$$

- *ABox sentences*, which assert memberships of individuals in concepts, written $a \in C$ for an individual $a \in \mathcal{I}$ and a concept C , or in roles, written $R(a, b)$ for individuals $a, b \in \mathcal{I}$ and a role $R \in \mathcal{R}$.

Sentence translation is defined inductively on the structure of the sentences by replacing atomic concepts, roles and individuals according to the corresponding mappings in the signature morphism. A model I consists of a set Δ^I called the domain or the universe and an interpretation function which assigns to each atomic concept C a unary predicate C^I on Δ^I (which can be identified to a subset of the domain where the respective predicate holds), to each role R a binary predicate R^I on Δ^I and to each individual a an element $a^I \in \Delta^I$. To define the satisfaction relation, we first extend the interpretation of atomic concepts to interpretation of arbitrary concepts, inductively on the structure, as follows:

$$\begin{aligned} \top^I &= \Delta^I \\ \perp^I &= \emptyset \\ (\neg C)^I &= \Delta^I \setminus C^I \\ (C_1 \sqcup C_2)^I &= C_1^I \cup C_2^I \\ (C_1 \sqcap C_2)^I &= C_1^I \cap C_2^I \\ (\forall R.C)^I &= \{x \in \Delta^I \mid \forall y \in \Delta^I. (x, y) \in R^I \text{ implies } y \in C^I\} \\ (\exists R.C)^I &= \{x \in \Delta^I \mid \exists y \in \Delta^I. (x, y) \in R^I\} \end{aligned}$$

Then a model I satisfies a subsumption relation $C_1 \sqsubseteq C_2$ if $C_1^I \subseteq C_2^I$, a membership assertion $a \in C$ if $a^I \in C^I$ and a role assertion $R(a, b)$ if $R^I(a^I, b^I)$ holds.

Example 3.1.8 (The institution of a programming language.) [Tarlecki, 1996] The institution **PLNG** of a programming language is built over an algebra of built-in data types and operations of a programming language. Signatures are given as function (functional procedure) headings; sentences are function bodies; and models are maps that for each function symbol, assign a computation (either diverging, or yielding a result) to any sequence of actual parameters. A model satisfies a sentence iff it assigns to each sequence of parameters the computation of the function body as given by the sentence. Hence, sentences determine particular functions in the model uniquely. Finally, signature morphisms, model reductions and sentence translations are defined similarly to those in **FOL**_{ms}.

We now give several institution-independent logical notions that will be used in the following chapters.

Let us fix an arbitrary institution $I = (\mathbf{Sig}, \mathbf{Sen}, \mathbf{Mod}, \models)$. We assume that the category of signatures has unions: for each two signatures Σ_1 and Σ_2 there is a signature Σ called the union of Σ_1 and Σ_2 and denoted $\Sigma_1 \cup \Sigma_2$ and two “injection” morphisms $\iota_{\Sigma_i, \Sigma} : \Sigma_i \rightarrow \Sigma$, $i = 1, 2$. This generalizes to arbitrary finite unions in the expected way. In practice, union of signatures may fail to be a total operation,

as it is for example the case when overloading is not a feature of the logic and two signatures give the same name to different things. We moreover assume that this operation has the usual properties of the set-theoretic union. In practice, this can be achieved by replacing institutions with the less abstract notion of *institution with qualified symbols* [Mossakowski, 2000] or by working with signature categories having associated an *inclusion system* [Căzănescu and Roşu, 1997]. Moreover, in a similar setting, assume that $\sigma_1 : \Sigma_1 \rightarrow \Sigma'_1$ and $\sigma_2 : \Sigma_2 \rightarrow \Sigma'_2$ are signature morphisms. Their union $\sigma_1 \cup \sigma_2 : \Sigma_1 \cup \Sigma_2 \rightarrow \Sigma'_1 \cup \Sigma'_2$ is defined as the unique morphism that makes the following diagram commute:

$$\begin{array}{ccc}
 \Sigma_1 & \xrightarrow{\sigma_1} & \Sigma'_1 \\
 \searrow & & \searrow \\
 & \Sigma_1 \cup \Sigma_2 & \xrightarrow{\sigma_1 \cup \sigma_2} & \Sigma'_1 \cup \Sigma'_2 \\
 \nearrow & & \nearrow \\
 \Sigma_2 & \xrightarrow{\sigma_2} & \Sigma'_2
 \end{array}$$

when such a signature morphism exists² and is undefined otherwise. This generalized to finite unions of signature morphisms in the expected way.

We can naturally extend the satisfaction relation to a relation between models and sets of sentences as follows: if Σ is a signature, M is a Σ -model and E is a set of Σ -sentences, we say that M is a model of E , denoted $M \models E$, if $M \models e$ for each $e \in E$. The next definition introduces the concept of semantical entailment between sentences.

Definition 3.1.9 (Logical consequence) *If Σ is a signature, E is a set of Σ -sentences and e is a Σ -sentence, we say that e is a logical consequence of E , written $E \models_{\Sigma} e$, if for all Σ -models we have that $M \models_{\Sigma} E$ implies $M \models_{\Sigma} e$.*

We omit the index when the signature is clear from the context.

A *theory* is a pair (Σ, E) where Σ is a signature and E is a set of Σ -sentences. A model of a theory (Σ, E) is a Σ -model M such that $M \models E$. Given two theories (Σ, E) and (Σ', E') , a theory morphism $\sigma : (\Sigma, E) \rightarrow (\Sigma', E')$ is a signature morphism $\sigma : \Sigma \rightarrow \Sigma'$ such that $E' \models \sigma(E)$. For a theory (Σ, E) , we denote $E^{\bullet} = \{e \in \mathbf{Sen}(\Sigma) \mid E \models_{\Sigma} e\}$.

Example 3.1.10 (Institution of theories) *Given an institution I , we define the institution of I -theories, denoted $I^{th} = (\mathbf{Sig}^{th}, \mathbf{Sen}^{th}, \mathbf{Mod}^{th}, \models^{th})$ as follows:*

- \mathbf{Sig}^{th} has as objects all I -theories and as arrows I -theory morphisms;
- for a theory (Σ, E) , $\mathbf{Mod}^{th}(\Sigma, E)$ consists of all Σ -models that satisfy E ;
- \mathbf{Sen}^{th} and \models^{th} are inherited from I .

²See [Mosses, 2004] for more details.

Definition 3.1.11 Given an institution $I = (\text{Sig}, \text{Sen}, \text{Mod}, \models)$, a theory morphism $\varphi : (\Sigma, E) \rightarrow (\Sigma', E')$ is called:

- model-theoretically conservative if any (Σ, E) -model M has (at least) an expansion along φ to a (Σ', E') -model, i.e. there is a model M' that satisfies E' such that $M' \upharpoonright_{\varphi} = M$;
- consequence-theoretically conservative if $E' \models \varphi(e)$ implies $E \models e$, for any Σ -sentence e .

It is known that the model-theoretic implies consequence-theoretic conservativity, but the converse is not true in general (see [Lutz et al., 2007] for an example involving description logics).

3.1.1 Colimits and Amalgamation

Colimits are a mean of combining interconnected objects consistently to this interconnection. They can be employed for constructing larger theories from already available smaller ones, see [Goguen and Burstall, 1992].

A *diagram* in a category C is a functor $D: G \rightarrow C$, where G is a small category, and can be thought of as the graph of interconnections between the objects of C selected by the functor D . A *cocone* of a diagram $D: G \rightarrow C$ consists of an object c of C and a family of morphisms $\alpha_i: D(i) \rightarrow c$, for each object i of G , such that for each edge of the diagram, $e: i \rightarrow i'$ we have that $D(e); \alpha_{i'} = \alpha_i$. A *colimiting cocone* (or *colimit*) $(c, \{\alpha_i\}_{i \in |G|})$ can be intuitively understood as a minimal cocone, i.e. has the property that for any cocone $(d, \{\beta_i\}_{i \in |G|})$ there exists a unique morphism $\gamma: c \rightarrow d$ such that $\alpha_i; \gamma = \beta_i$. By dropping the uniqueness condition and requiring only that a morphism γ should exist, we obtain a *weak colimit*.

When G is the category $\bullet \leftarrow \bullet \rightarrow \bullet$ with 3 objects and 2 non-identity arrows, the G -colimits are called *pushouts*.

Since specifications are actually *theories* over some institution (i.e. pairs (Σ, E) with Σ a signature and E a set of Σ -sentences) we are actually interested in computing colimits of theories rather than just signatures. A result in [Goguen and Burstall, 1992] ensures that to obtain a colimit of theories, it suffices to compute the colimit of signatures and then the set of sentences of the colimit theory is defined as the union of all component theories in the diagram, translated along the signature morphisms of the colimiting cocone.

A major property of colimits of specifications is *amalgamation* (called ‘exactness’ in [Diaconescu et al., 1993]). It can be intuitively explained [Schröder et al., 2005] as stating that models of given specifications can be combined to yield a uniquely determined model of a colimit specification, provided that the original models coincide on common components. Amalgamation is a common technical assumption in the study of specification semantics [Sannella and Tarlecki, 1988b]. For example, computation of normal forms for specifications [Borzyszkowski, 2002], semantics of parametrization and conservative extensions [Diaconescu et al., 1993] require

amalgamation property for the underlying institution. The proof system for development graphs with hiding (Sec. 4.3) is sound only for logics with amalgamation.

In the sequel, fix an arbitrary institution $I = (\mathbf{Sig}, \mathbf{Sen}, \mathbf{Mod}, \models)$.

Definition 3.1.12 *Given a diagram $D: J \longrightarrow \mathbf{Sig}^I$, a family of models $\mathcal{M} = \{M_p\}_{p \in |J|}$ is consistent with D (or sometimes compatible with D) if for each node p of D , $M_p \in \mathbf{Mod}(D(p))$ and for each edge $e: p \rightarrow q$, $M_p = M_q|_{D(e)}$. A cocone $(\Sigma, (\mu_j)_{j \in |J|})$ over the diagram in $D: J \longrightarrow \mathbf{Sig}^I$ is called weakly amalgamable if it is mapped to a weak limit by \mathbf{Mod} . For models, this means that for each D -compatible family of models $(M_j)_{j \in |J|}$, there is a Σ -model M with $M \downarrow_{\mu_j} = M_j$ ($j \in |J|$), and similarly for model morphisms. If this model is unique, the cocone is called amalgamable. I (or \mathbf{Mod}) admits (finite) (weak) amalgamation if (finite) colimit cocones are (weakly) amalgamable. Finally, I is called (weakly) semi-exact if it has pushouts and admits (weak) amalgamation for these. \square*

3.1.2 Institutions with Proofs

The logical consequence relation \models provides a semantic way of establishing truth: a sentence e is a logical consequence of a set of sentences E if all models that satisfy E also satisfy e . This is often too inefficient to use, as the models we have to consider may be infinitely many, and therefore a syntactical approach is preferred, by giving a proof of the sentence or sentences we want to establish true from a set of assumptions by applying syntactic transformations. The latter are usually given in the form of a calculus, which consists of a set of syntactic rules.

The notion of entailment system (well known in proof theory, but first introduced in the context of institutions in [Meseguer, 1989]) provides an abstract view on calculi, allowing to formulate general properties.

Definition 3.1.13 *Given an institution $I = (\mathbf{Sig}, \mathbf{Sen}, \mathbf{Mod}, \models)$, an entailment system for I is a relation $\vdash_\Sigma \subseteq \mathcal{P}(\mathbf{Sen}(\Sigma)) \times \mathbf{Sen}(\Sigma)$ for each signature Σ , such that:*

1. reflexivity: for any $e \in \mathbf{Sen}(\Sigma)$, $\{e\} \vdash_\Sigma e$;
2. monotonicity: if $E \vdash_\Sigma e$ and $E \subseteq E'$, then $E' \vdash_\Sigma e$;
3. transitivity: if $E \vdash_\Sigma e_i$, for $i \in \mathit{Ind}$ and $E \cup \{e_i\}_{i \in \mathit{Ind}} \vdash_\Sigma e$, then $E \vdash_\Sigma e$;
4. translation: if $E \vdash_\Sigma e$ and $\sigma: \Sigma \rightarrow \Sigma'$, then $\sigma(E) \vdash_{\Sigma'} \sigma(e)$;

Moreover, an entailment system for I is:

- sound if $E \vdash_\Sigma e$ implies $E \models_\Sigma e$;
- complete if $E \models_\Sigma e$ implies $E \vdash_\Sigma e$.

Note that in HETS entailment systems are not represented explicitly, as they are rather a part of the tools that are interfaced by HETS. In Chapter 6 we discuss an extension of HETS which also allows to represent the proof theory of a logic.

Conservativity of signature morphisms can be given a proof theoretical meaning as well.

Definition 3.1.14 *Given an institution $I = (\mathbf{Sig}, \mathbf{Sen}, \mathbf{Mod}, \models)$ and an entailment system \vdash_Σ for I , a theory morphism $\varphi : (\Sigma, E) \rightarrow (\Sigma', E')$ is proof-theoretically conservative if $E' \vdash_{\Sigma'} \varphi(e)$ implies $E \vdash_\Sigma e$, for any Σ -sentence e .*

3.2 Institution Comorphisms

In heterogeneous specification, we are interested in combining specifications that are written in different logics. To support this, several types of translations between institutions have been introduced. Among them, institution comorphisms [Goguen and Roşu, 2002] typically express that an institution is included or encoded into another one. An institution comorphism provides

- a translation of signatures (and signature morphisms),
- a translation of sentences,
- a translation of models (going backwards to the direction of the comorphism),

such that satisfaction is invariant under translation of sentences resp. models.

Definition 3.2.1 *Given two institutions $I_1 = (\mathbf{Sig}_1, \mathbf{Sen}_1, \mathbf{Mod}_1, \models_1)$ and $I_2 = (\mathbf{Sig}_2, \mathbf{Sen}_2, \mathbf{Mod}_2, \models_2)$, an institution comorphism consists of a functor $\varphi : \mathbf{Sig}_1 \rightarrow \mathbf{Sig}_2$, a natural transformation $\beta : \varphi; \mathbf{Mod}_2 \Rightarrow \mathbf{Mod}_1$ and a natural transformation $\alpha : \mathbf{Sen}_1 \rightarrow \varphi; \mathbf{Sen}_2$ such that the following satisfaction condition holds for each $\Sigma \in |\mathbf{Sig}_1|$, $M' \in |\mathbf{Mod}_2(\varphi(\Sigma))|$ and $e \in \mathbf{Sen}_1(\Sigma)$*

$$\beta_\Sigma(M') \models_\Sigma e \Leftrightarrow M' \models_{\varphi(\Sigma)} \alpha_\Sigma(e)$$

Example 3.2.2 *The obvious inclusions from \mathbf{FOL}_{ms} to \mathbf{PFOL}_{ms} and from \mathbf{FOL}_{ms} to \mathbf{HOL}_{ms} are institution comorphisms.*

Example 3.2.3 *The translation of \mathcal{ALC} to \mathbf{FOL}_{ms} [Baader et al., 2003] can be defined as follows. Each \mathcal{ALC} -signature $\Sigma = (\mathcal{A}, \mathcal{R}, \mathcal{I})$ is translated to a signature $\Phi(\Sigma) = (S, F, P)$ where $S = \{\text{Thing}\}$, $F = \{a : \text{Thing} \mid a \in \mathcal{I}\}$ and $P = \{C : \text{Thing} \mid C \in \mathcal{A}\} \cup \{R : \text{Thing} \times \text{Thing} \mid R \in \mathcal{R}\}$. An \mathcal{ALC} -signature morphism σ is translated to the signature morphism that maps constants according to $\sigma^\mathcal{I}$, unary predicates according to $\sigma^\mathcal{A}$ and binary predicates according to $\sigma^\mathcal{R}$.*

For an \mathcal{ALC} -signature $\Sigma = (\mathcal{A}, \mathcal{R}, \mathcal{I})$, the sentence translation function α_Σ is based on translation of concepts α_x :

$$\alpha_\Sigma(C_1 \sqsubseteq C_2) = \forall x : \text{Thing}(\alpha_x(C_1) \implies \alpha_x(C_2))$$

$$\alpha_\Sigma(a : C) = \alpha_x(C)[a/x]$$

$$\alpha_\Sigma(R(a, b)) = R(a, b)$$

where α_x is defined as follows for a variable $x : \text{Thing}$:

$$\alpha_x(A) = A(x)$$

$$\alpha_x(\top) = \text{True}$$

$$\alpha_x(\perp) = \text{False}$$

$$\alpha_x(\neg C) = \neg \alpha_x(C)$$

$$\alpha_x(C_1 \sqcup C_2) = \alpha_x(C_1) \vee \alpha_x(C_2)$$

$$\alpha_x(C_1 \sqcap C_2) = \alpha_x(C_1) \wedge \alpha_x(C_2)$$

$$\alpha_x(\forall R.C) = \forall y : \text{Thing}(R(x, y) \implies \alpha_y(C))$$

$$\alpha_x(\exists R.C) = \exists y : \text{Thing}(R(x, y) \wedge \alpha_y(C))$$

Given an \mathcal{ALC} -signature $\Sigma = (\mathcal{A}, \mathcal{R}, \mathcal{I})$ and a $\Phi(\Sigma)$ -model M , we define $\beta_\Sigma(M) = (\Delta, \cdot^I)$ where $\Delta = M_{\text{Thing}}$, $A^I = M_A$ for each atomic concept A , $a^I = M_a$ for each individual a and $R^I = M_R$ for each role R . \square

Example 3.2.4 The comorphism from the institution of relational schemes \mathbf{Rel} to \mathbf{FOL}_{ms} maps signatures and models in the straightforward way, while each \mathbf{Rel} -sentence $R(f_1, \dots, f_n) \rightarrow R'(f'_1, \dots, f'_m)$ linking fields f_i and f'_j is translated to the first-order formula $R(x_1, \dots, x_n) \implies \exists y_1 \dots y_{j-1} y_{j+1} \dots y_m R'(y_1, \dots, y_{j-1}, x_i, y_{j+1}, \dots, y_m)$. \square

Example 3.2.5 There is an institution comorphism from \mathbf{PFOL}_{ms} to \mathbf{FOL}_{ms}^{th} that codes out partiality via error elements. Details can be found in [Mossakowski, 2002b]. By composing with the inclusion from \mathbf{FOL}_{ms}^{th} to \mathbf{HOL}_{ms}^{th} , we get a comorphism from \mathbf{PFOL}_{ms} to \mathbf{HOL}_{ms}^{th} . \square

Example 3.2.6 There is an institution comorphism from \mathbf{PLNG} to \mathbf{HOL}_{ms}^{th} that codes the semantics of \mathbf{PLNG} within higher-order logic. \square

Institution morphisms [Goguen and Roşu, 2002] are another important class of translations between institutions, usually capturing projections from a more expressive institution to a less expressive one. The definition is similar to the one of institution comorphisms, only this time sentences are translated against and models in the direction of the institution morphism. Finally, institution semi-morphisms and institution semi-comorphisms differ from morphisms and comorphisms respectively by that they do not have a sentence translation component, and thus no satisfaction condition.

Example 3.2.7 *There is an institution semi-morphism $toPFOL$ from $PLNG$ to $PFOL_{ms}$ [Tarlecki, 1996]. It extracts an algebraic signature $\Phi(\Sigma)$ with partial operations out of a $PLNG$ -signature Σ by adding the signature of built-in data types and operations of the programming language. For any function declared, any $PLNG$ -model M determines its computations on given arguments, from which we can extract a partial function that maps any sequence of arguments to the result of the computation (if any). These are used to expand the built-in algebra of data types and operations of the programming language with an interpretation for the extra function names in the signature obtained, thus obtaining a $PFOL_{ms}$ -model $\beta(M)$.*

In our setting, this can be modelled as a span of comorphisms (cf. [Mossakowski, 2003])

$$PLNG \xleftarrow{toPFOL^-} \Phi; PFOL_{ms} \xrightarrow{toPFOL^+} PFOL_{ms}$$

as follows:

$$\begin{array}{ccccc} \mathbf{Sig}^{PLNG} & \xleftarrow{id} & \mathbf{Sig}^{PLNG} & \xrightarrow{\Phi} & \mathbf{Sig}^{PFOL} \\ \mathbf{Sen}^{PLNG} & \xleftarrow{incl} & \emptyset & \xrightarrow{incl} & \Phi; \mathbf{Sen}^{PFOL} \\ \mathbf{Mod}^{PLNG} & \xrightarrow{\beta} & \Phi^{op}; \mathbf{Mod}^{PFOL_{ms}} & \xleftarrow{id} & \Phi^{op}; \mathbf{Mod}^{PFOL} \end{array}$$

Here, the “middle” institution $PFOL_{ms} \circ \Phi$ is the institution with signature category inherited from $PLNG$, no sentences, and models inherited from $PFOL_{ms}$ via Φ . \square

Definition 3.2.8 *An institution comorphism is model-expansive, if each model translation β_Σ is surjective on objects.*

All of the comorphisms above, except the second one from Example 3.2.7, are model expansive. Model-expansive comorphisms allow for translating questions about logical consequence from I into J , which leads to the possibility of re-using a proof system for J also for I :

Proposition 3.2.9 *Given a model-expansive comorphism $(\Phi, \alpha, \beta): I \longrightarrow J$,*

$$\Gamma \models^I \varphi \text{ iff } \alpha_\Sigma(\Gamma) \models^J \alpha(\varphi)$$

Amalgamation resp. exactness can be lifted to comorphisms as follows:

Definition 3.2.10 *Let $\rho = (\Phi, \alpha, \beta): I \longrightarrow J$ be an institution comorphism and let \mathcal{D} be a class of signature morphisms in I . Then ρ is said to have the (weak) \mathcal{D} -amalgamation property, if for each signature morphism $\sigma: \Sigma_1 \longrightarrow \Sigma_2 \in \mathcal{D}$, the diagram*

$$\begin{array}{ccc} \mathbf{Mod}^I(\Sigma_2) & \xleftarrow{\beta_{\Sigma_2}} & \mathbf{Mod}^J(\Phi(\Sigma_2)) \\ \mathbf{Mod}^I(\sigma) \downarrow & & \downarrow \mathbf{Mod}^J(\Phi(\sigma)) \\ \mathbf{Mod}^I(\Sigma_1) & \xleftarrow{\beta_{\Sigma_1}} & \mathbf{Mod}^J(\Phi(\Sigma_1)) \end{array}$$

admits (weak) amalgamation, i.e. for any two models $M_2 \in \mathbf{Mod}^I(\Sigma_2)$ and $M'_1 \in \mathbf{Mod}^J(\Phi(\Sigma_1))$ with $M_2 \upharpoonright \sigma = \beta_{\Sigma_1}(M'_1)$, there is a unique (not necessarily unique) $M'_2 \in \mathbf{Mod}^J(\Phi(\Sigma_2))$ with $\beta_{\Sigma_2}(M'_2) = M_2$ and $M'_2 \upharpoonright \Phi(\sigma) = M'_1$. In case that \mathcal{D} consists of all signature morphisms, the (weak) \mathcal{D} -amalgamation property is also called (weak) exactness. \square

3.3 Grothendieck Institutions

The Grothendieck construction for indexed institutions (based on institution morphisms) has been defined in [Diaconescu, 2002]; we here describe the dual, comorphism-based variant [Mossakowski, 2002a]. The idea is to begin with a graph of logics and logics translations formalized as comorphisms and then to flatten this graph, using a so-called Grothendieck construction. This construction is characterized by the fact that no interaction between logics is made otherwise than via the logic translations.

Definition 3.3.1 Given an index category Ind , an indexed coinstitution is a functor $\mathcal{I}: Ind^{op} \rightarrow \mathbf{CoIns}$ into the category of institutions and institution comorphisms.

In an indexed coinstitution \mathcal{I} , we use the notations $\mathcal{I}^i = (\mathbf{Sig}^i, \mathbf{Sen}^i, \mathbf{Mod}^i, \models^i)$ for $\mathcal{I}(i)$ and $(\Phi^d, \alpha^d, \beta^d)$ for the comorphism $\mathcal{I}(d)$.

Definition 3.3.2 Given an indexed coinstitution $\mathcal{I}: Ind^{op} \rightarrow \mathbf{CoIns}$, we define the Grothendieck institution $\mathcal{I}^\# = (\mathbf{Sig}^\#, \mathbf{Sen}^\#, \mathbf{Mod}^\#, \models^\#)$ as follows:

- signatures in $\mathcal{I}^\#$ are pairs (i, Σ) , where $i \in |Ind|$ and Σ a signature in \mathcal{I}^i ,
- signature morphisms $(d, \sigma): (i, \Sigma_1) \rightarrow (j, \Sigma_2)$ consist of a morphism $d: j \rightarrow i \in Ind$ and a signature morphism $\sigma: \Phi^d(\Sigma_1) \rightarrow \Sigma_2$ in \mathcal{I}^j ,
- if $(d_1, \sigma_1): (i, \Sigma) \rightarrow (j, \Sigma')$ and $(d_2, \sigma_2): (j, \Sigma') \rightarrow (k, \Sigma'')$, the composition $(e, \sigma) := (d_1, \sigma_1); (d_2, \sigma_2): (i, \Sigma) \rightarrow (k, \Sigma'')$ is obtained by defining e as the composition of d_2 and d_1 :

$$k \xrightarrow{d_2} j \xrightarrow{d_1} i$$

$$\searrow e \nearrow$$

and σ as the composition of $\Phi^{d_2}(\sigma_1)$ and σ_2 :

$$\Phi^{d_2}(\Phi^{d_1}(\Sigma)) \xrightarrow{\Phi^{d_2}(\sigma_1)} \Phi^{d_2}(\Sigma') \xrightarrow{\sigma_2} \Sigma''$$

$$\searrow \sigma \nearrow$$

- for each signature (i, Σ) ,

$$\mathbf{Sen}^\#(i, \Sigma) = \mathbf{Sen}^i(\Sigma)$$

and

$$\mathbf{Mod}^\#(i, \Sigma) = \mathbf{Mod}^i(\Sigma)$$

Moreover $M \models^\# e$ iff $M \models^i e$ for any Σ -model M and any Σ -sentence e ;

- for each signature morphism $(d, \sigma): (i, \Sigma_1) \rightarrow (j, \Sigma_2)$, the sentence translation function $\mathbf{Sen}^\#(d, \sigma): \mathbf{Sen}^i(\Sigma_1) \rightarrow \mathbf{Sen}^j(\Sigma_2)$ is defined by first translating the sentence along $\mathcal{I}(d)$ and then along the morphism σ :

$$\mathbf{Sen}^i(\Sigma_1) \xrightarrow{\alpha_{\Sigma_1}^d} \mathbf{Sen}^j(\Phi^d(\Sigma_1)) \xrightarrow{\mathbf{Sen}^j(\sigma)} \mathbf{Sen}^j(\Sigma_2)$$

while the model reduct $\mathbf{Mod}^\#(d, \sigma): \mathbf{Mod}^j(\Sigma_2) \rightarrow \mathbf{Mod}^i(\Sigma_1)$ is defined by first reducing along the signature morphism and then along $\mathcal{I}(d)$:

$$\mathbf{Mod}^j(\Sigma_2) \xrightarrow{\mathbf{Mod}^j(\sigma)} \mathbf{Mod}^j(\Phi^d(\Sigma_1)) \xrightarrow{\beta_{\Sigma_1}} \mathbf{Mod}^i(\Sigma_1)$$

That is, sentences, models and satisfaction for a Grothendieck signature (i, Σ) are defined component wise, while the sentence and model translations for a Grothendieck signature morphism are obtained by composing the translation given by the inter-institution comorphism with the one given by the intra-institution signature morphism.

[Mossakowski and Tarlecki, 2009] introduce the notion of *heterogeneous logical environment* as an alternative framework for heterogeneous specification. The difference is that a *distributed* approach is pursued, in the sense that the specifications are not integrated to a unique institution. Instead, a diagram of specifications, possibly in different formalisms, forms a single distributed specification.

Contents

4.1 CASL Logic	45
4.2 Structured Specifications	47
4.3 Development Graphs	51
4.4 CASL Architectural Specifications	53
4.5 HETCASL	58

CASL is a general-purpose algebraic specification language. We give here an overview of its layers, starting with the CASL logic for basic specifications together with some syntactical constructs to explain the specifications presented throughout this thesis, in Sec. 4.1. CASL provides then structuring mechanisms (Section 4.2) that allow to compose specifications, making them easier to understand, maintain and reuse. CASL uses the formalism of development graphs (Section 4.3) for structured theorem proving, proof management and management of change. As a counterpart to structured specifications, architectural specifications (Section 4.4) provide means for structuring *implementations* of modular systems. Finally, HETS extends the structuring mechanism of CASL at the heterogeneous level (Section 4.5), the resulting language being called HETCASL.

4.1 CASL Logic

The logic used at the basic level of CASL consists of an extension of first-order logic with a combination of features commonly used in specification, like subsorting, partiality or induction axioms. The resulting logic is many-sorted partial first-order logic with equality, induction and subsorting, also denoted SubPCFOL_{ms} . We use SubPCFOL_{ms} and *CASL* to denote the same logic. Note that CASL also provides notational means for selecting only some of these features for the logic currently in use, thus resulting sublogics and sublanguages of CASL.

CASL signatures consists of a set of sorts with a subsort relation between sorts, denoted \leq , a set of partial function symbols such that some of them are marked as total and a set of predicate symbols. Moreover, if Σ is a signature, two operation symbols with the same name f and with profiles $w \rightarrow s$ and $w' \rightarrow s'$, denoted $f_{w,s}$ and $f_{w',s'}$ to disambiguate, are in the overloading relation, denoted $f_{w,s} \sim_F f_{w',s'}$ if there are $w_0 \in S^*$ and $s_0 \in S$ such that $w_0 \leq w, w'$ and $s, s' \leq s_0$. Overloading of

predicates is defined in a similar way. Signature morphisms consist of maps taking sort, function and predicate symbols respectively to a symbol of the same kind in the target signature, and they must preserve subsorting, typing of function and predicate symbols, totality of function symbols and overloading.

For a signature $\Sigma = (S, TF, PF, P)$, terms with variables from a sorted, non-overloaded set X are formed with variables from X and applications of (total or partial) function symbols to terms of appropriate sorts, while sentences are partial first-order formulas, defined inductively as follows:

- the atomic formulas are application of predicate symbols to a list of terms of appropriate sorts, assertion about definedness of terms, existential and strong equality between terms of the same sort;
- if X is a sorted set of variables and Φ, Φ_1, Φ_2 are formulas, then $\neg\Phi, \Phi_1 \wedge \Phi_2, \Phi_1 \vee \Phi_2, \Phi_1 \implies \Phi_2, \Phi_1 \iff \Phi_2, \forall X \bullet \Phi$ and $\exists X \bullet \Phi$ are formulas.

Moreover, $\mathbf{Sen}(\Sigma)$ includes *sort generation constraints*: a sort generation constraint is a triple (S', F', σ') such that $\sigma' : \Sigma' \rightarrow \Sigma$ and S' and F' are respectively sort and function symbols of Σ' . Sort generation constraints are translated along a signature morphism $\varphi : \Sigma \rightarrow \Sigma''$ by composing the morphism σ' in their third component with φ .

Models interpret sorts as sets such that subsorts are injected into supersorts, partial/total function symbols as partial/total functions and predicate symbols as relations such that the embeddings of subsorts into supersorts are monotone w.r.t. overloading.

The satisfaction relation is the expected one for partial first-order sentences. A sort generation constraint (S', F', σ') holds in a model M if the carriers of $M|_{\sigma'}$ of the sorts in S' are generated by function symbols in F' .

Theorem 4.1.1 *CASL is an institution.*

A sound proof calculus for the CASL logic was introduced in [Mosses, 2004] and [Mossakowski et al., 2008]. Completeness can be obtained only when sort generation constraints are not used.

We now present the syntactic constructs for writing basic specifications, in a very informal manner. [Mosses, 2004] gives a complete description of the CASL language grammar. A CASL specification uses the keyword **sort** for sorts, **<** for subsorting, the keyword **op** for function symbols and **->?** in the profile of a symbol to denote its partiality, and finally the keyword **pred** for predicate symbols. If more than a symbol is declared, the keywords appear in plural and the declarations are separated by commas. Binary operations may additionally have an attribute for declaring them associative, commutative, idempotent or with a unit. Mixfix notation is allowed, using the placeholders **__**. Sort generation constraints are expressed concisely using the **generated type** and **free type** constructions. Finally, axioms are prefixed with

a bullet, and the annotation `%implied` is used to mark asserted theorems. Note that these last two syntactic constructions are institution-indepedent.

A basic specification denotes a signature and a class of models. The static semantics rules, of form $\vdash SP \triangleright \Sigma$, check whether a specification is syntactically correct and produce a signature Σ as result when that is indeed the case. If the static semantics succeeds, the model semantics rules, of form $\vdash SP \Rightarrow \mathcal{M}$, check the semantic correctness of the specification and produce a class \mathcal{M} of Σ -models. If any of the analyses fails, the specification does not have a denotation. Note that the semantics is *loose*, that is, all models that satisfy the axioms of a specification will be in the model class produced by the model semantics analysis.

4.2 Structured Specifications

The simplest specifications are the so-called basic specifications, obtained by pairing a signature and a set of sentences over that signature. Working with basic specifications is only suitable for specifications of fairly small size. Formal specification is however usually employed for large systems. Therefore, for practical situations, unstructured specifications would become impossible to understand and use efficiently. Moreover, a modular design allows for reuse of specifications.

In the following we will present a kernel of structuring-building operations over an arbitrary institution I , very similar to the ones of CASL and those introduced in [Sannella and Tarlecki, 1988b]. We adhere to the model-theoretic semantics, i.e. a specification denotes a signature and a class of models over that signature. Thus, the signature of a structured specification SP will be given by the static semantics rules, for proving judgements of the form $\vdash SP \triangleright \Sigma$, where Σ is a signature and then we sometimes denote it by $\mathbf{Sig}(SP)$. Similarly, the class of models of SP will be given by the model semantics rules, with judgements of the form $\vdash SP \Rightarrow \mathcal{M}$ where \mathcal{M} is a class of Σ -models and then we sometimes denote it by $\mathbf{Mod}(SP)$.

- **hiding:**

$$\frac{\vdash SP' \triangleright \Sigma' \quad \sigma : \Sigma \rightarrow \Sigma'}{\vdash SP \mathbf{hide} \sigma \triangleright \Sigma}$$

and

$$\frac{\vdash SP' \Rightarrow \mathcal{M} \quad \mathcal{M}' = \{M' |_{\sigma} \mid M' \in \mathcal{M}\}}{\vdash SP' \mathbf{hide} \sigma \Rightarrow \mathcal{M}'}$$

The simplest form of refinement is model class inclusion between structured specifications: if $\sigma : \Sigma_1 \rightarrow \Sigma_2$ and SP_1, SP_2 are structured specifications such that $\vdash SP_i \triangleright \Sigma_i$ and $\vdash SP_i \Rightarrow \mathcal{M}_i$, $i = 1, 2$, we say that SP_1 refines along σ to SP_2 , denoted $SP_1 \rightsquigarrow_{\sigma} SP_2$ if $M_2 |_{\sigma} \in \mathcal{M}_1$ for all $M_2 \in \mathcal{M}_2$. CASL already allows to postulate such refinements using *views*: if SP_1 and SP_2 are structured

specifications, then **view** $V : SP_1 \text{ to } SP_2 = \sigma$ requires that each model of SP_2 reduced along σ should be a model of SP_1 .

Thus, the signature of a structured specification SP will be given by the static semantics rules, of form $\vdash SP \triangleright \Sigma$ where Σ is a signature, while the class of models of SP will be given by the model semantics rules, of form $\vdash SP \Rightarrow \mathcal{M}$ where \mathcal{M} is a class of Σ -models which we sometimes denote $\mathbf{Mod}(SP)$.

Presentations: Given a signature Σ and a set of Σ -sentences E , (Σ, E) is a structured specification such that:

$$\vdash (\Sigma, E) \triangleright \Sigma$$

and

$$\vdash (\Sigma, E) \Rightarrow \mathbf{Mod}_\Sigma(E)$$

Union: For two specifications SP_1 and SP_2 , $SP_1 \text{ and } SP_2$ is a structured specification such that

$$\frac{\begin{array}{l} \vdash SP_1 \triangleright \Sigma_1 \\ \vdash SP_2 \triangleright \Sigma_2 \end{array}}{\vdash SP_1 \text{ and } SP_2 \triangleright \Sigma_1 \cup \Sigma_2}$$

and

$$\frac{\begin{array}{l} \vdash SP_1 \Rightarrow \mathcal{M}_1 \\ \vdash SP_2 \Rightarrow \mathcal{M}_2 \\ \mathcal{M} = \{M \in \mathbf{Mod}(\Sigma_1 \cup \Sigma_2) \mid M|_{\iota_{\Sigma_i \subseteq \Sigma_1 \cup \Sigma_2}} \in \mathcal{M}_i, i = 1, 2\} \end{array}}{\vdash SP_1 \text{ and } SP_2 \Rightarrow \mathcal{M}}$$

Translation: $SP \text{ with } \sigma$ is a structured specification such that

$$\frac{\begin{array}{l} \vdash SP \triangleright \Sigma \\ \sigma : \Sigma \rightarrow \Sigma' \end{array}}{\vdash SP \text{ with } \sigma \triangleright \Sigma'}$$

and

$$\frac{\begin{array}{l} \vdash SP \Rightarrow \mathcal{M} \\ \mathcal{M}' = \{M' \in \mathbf{Mod}(\Sigma') \mid M'|_\sigma \in \mathcal{M}\} \end{array}}{\vdash SP \text{ with } \sigma \Rightarrow \mathcal{M}'}$$

Hiding: $SP' \text{ hide } \sigma$ is a structured specification such that

$$\frac{\begin{array}{l} \vdash SP' \triangleright \Sigma' \\ \sigma : \Sigma \rightarrow \Sigma' \end{array}}{\vdash SP' \text{ hide } \sigma \triangleright \Sigma}$$

and

$$\frac{\begin{array}{l} \vdash SP' \Rightarrow \mathcal{M} \\ \mathcal{M}' = \{M'|_\sigma \mid M' \in \mathcal{M}\} \end{array}}{\vdash SP' \text{ hide } \sigma \Rightarrow \mathcal{M}'}$$

Freeness: **free** SP' **along** σ is a structured specification such that

$$\frac{\begin{array}{c} \vdash SP' \triangleright \Sigma' \\ \sigma : \Sigma \rightarrow \Sigma' \end{array}}{\vdash \mathbf{free} SP' \mathbf{along} \sigma \triangleright \Sigma'}$$

and

$$\frac{\begin{array}{c} \vdash SP' \Rightarrow \mathcal{M} \\ G = \mathbf{Mod}(\sigma) : \mathbf{Mod}(SP') \rightarrow \mathbf{Mod}(\Sigma) \\ \mathcal{M}' = \{M \in \mathcal{M} \mid M' \text{ is strongly persistently } G - \text{free}\} \end{array}}{\vdash \mathbf{free} SP' \mathbf{along} \sigma \Rightarrow M'}$$

where we denoted the reduct functor with G to distinguish it from the reduct $_ \downarrow \sigma$ defined on the whole category $\mathbf{Mod}(\Sigma')$ and the strongly persistency condition is given by the following definition:

Definition 4.2.1 Let $F : \mathcal{A} \rightarrow \mathcal{B}$ be a functor. An object M of \mathcal{A} is strongly persistently F -free if for any model N of \mathcal{A} and any arrow $h : F(M) \rightarrow F(N)$ in \mathcal{B} , there is a unique arrow $h^\# : M \rightarrow N$ in \mathcal{A} such that $F(h^\#) = h$.

CASL additionally provides *extension of theories* using the keyword **then** and optionally an annotation that the resulting theory inclusion is conservative, monomorphic, definitional or implicational. We further omit local and closed specifications as they do not appear in our examples. Moreover, CASL specifications can be *generic*, using parameters that can be later instantiated. We refer the interested reader to [Mosses, 2004].

Note that in some cases (but not all) it is possible to "flatten" structured specifications to a basic specification presenting the same class of models. We discuss this in the context of development graphs, in next section.

We can extend logical consequence to a relation between structured specifications and sentences.

Definition 4.2.2 Let SP a Σ -specification and let $e \in \text{Sen}(\Sigma)$. Then $SP \models_\Sigma e$ (SP logically entails e) if $M \models e$ for any $M \in \mathbf{Mod}(SP)$.

Example 4.2.3 We give some simple examples of CASL structured specifications. First we give a basic specification for monoids by introducing a sort for elements and a binary operation which is associative and has a unit:

```
spec MONOID =
  sort Elem
  ops e : Elem;
  ___*___ : Elem × Elem → Elem, assoc, unit e
```

A group is then a monoid where each element has an inverse:

spec GROUPINV =
 MONOID
then op $inv : Elem \rightarrow Elem$
 $\forall x : Elem$
 • $inv(x) * x = e$
 • $x * inv(x) = e$

Finally, we can hide the inverse:

spec GROUP =
 GROUPINV **hide** inv

Simple refinements between structured specifications can already be expressed as so-called *views* in CASL, which denote morphisms of specifications: if $\sigma : \Sigma_1 \rightarrow \Sigma_2$ and SP_1, SP_2 are structured specifications such that $\vdash SP_i \triangleright \Sigma_i$ and $\vdash SP_i \Rightarrow \mathcal{M}_i$, $i = 1, 2$, $\sigma : SP_1 \rightarrow SP_2$ is a morphism of specifications if $M_2|_\sigma \in \mathcal{M}_1$ for all $M_2 \in \mathcal{M}_2$. Views are specified in HETS using the keyword **view** and then giving the source and target specifications followed by the signature morphism (as a symbol map, see details in [Mosses, 2004]). We assume that there exists a sound proof calculus for checking that such σ is indeed a morphism of specifications; possible choices include the calculus defined in [Sannella and Tarlecki, 2012] and the formalism of development graphs provided by HETS (see next section). We will also use the notation $SP \rightsquigarrow SP'$ to denote that SP' is a refinement of SP .

Finally, we can introduce constructs for translating a structured specification along an institution comorphism. This construction is standard in approaches to heterogeneous specification, see e.g. [Tarlecki, 1998].

Definition 4.2.4 Let $\rho = (\Phi, \alpha, \beta) : I \rightarrow J$ be an institution comorphism. For any I -specification SP with $\vdash SP \triangleright \Sigma$ and $\vdash SP \Rightarrow \mathcal{M}$, $\rho(SP)$ is a specification called the translation of SP along ρ , such that

$$\vdash \rho(SP) \triangleright \Phi(\Sigma)$$

and

$$\vdash \rho(SP) \Rightarrow \{M \in \mathbf{Mod}(\Phi(\Sigma)) \mid \beta_\Sigma(M) \in \mathcal{M}\}$$

The following definition gives conditions for a comorphism to translate a specification in a way that behaves well with entailment.

Definition 4.2.5 A comorphism $\rho = (\Phi, \alpha, \beta) : I \rightarrow J$

- admits borrowing of entailment if for any Σ -specification SP and any Σ -sentence e , $SP \models_\Sigma^I e \iff \rho(SP) \models_{\Phi(\Sigma)}^J \alpha_\Sigma(e)$;
- admits borrowing of refinement if $SP_1 \rightsquigarrow SP_2$ iff $\rho(SP_1) \rightsquigarrow \rho(SP_2)$.

4.3 Development Graphs

For proof management, CASL uses *development graphs* [Mossakowski et al., 2006a]. They can be defined over an arbitrary institution, and they are used to encode structured specifications in various phases of the development. Roughly speaking, each node of the graph represents a theory. The links of the graph define how theories can make use of other theories.

The following definitions (up to Def. 4.3.3) are all copied from [Mossakowski et al., 2006a].

Definition 4.3.1 A development graph is an acyclic, directed graph $\mathcal{DG} = \langle \mathcal{N}, \mathcal{L} \rangle$.

\mathcal{N} is a set of nodes. Each node $N \in \mathcal{N}$ is a tuple (Σ^N, Φ^N) such that Σ^N is a signature and $\Phi^N \subseteq \text{Sen}(\Sigma^N)$ is the set of **local axioms** of N .

\mathcal{L} is a set of directed links, so-called **definition links**, between elements of \mathcal{N} . Each definition link from a node M to a node N is either

- **global** (denoted $M \xrightarrow{\sigma} N$), annotated with a signature morphism $\sigma : \Sigma^M \rightarrow \Sigma^N$, or
- **local** (denoted $M \xrightarrow{\sigma} N$), again annotated with a signature morphism $\sigma : \Sigma^M \rightarrow \Sigma^N$, or
- **hiding** (denoted $M \xrightarrow[\text{hide}]{\sigma} N$), annotated with a signature morphism $\sigma : \Sigma^N \rightarrow \Sigma^M$ going against the direction of the link, or
- **free** (denoted $M \xrightarrow[\text{free}]{\sigma} N$), annotated with a signature morphism $\sigma : \Sigma \rightarrow \Sigma^M$ where Σ is a subsignature of Σ^M .

Definition 4.3.2 Given a node M in a development graph \mathcal{DG} , its associated class $\text{Mod}_{\mathcal{DG}}(M)$ of models (or M -models for short) is inductively defined to consist of those Σ^M -models m for which

1. m satisfies the local axioms Φ^M ,
2. for each $N \xrightarrow{\sigma} M \in \mathcal{DG}$, $m|_{\sigma}$ is an N -model,
3. for each $N \xrightarrow{\sigma} M \in \mathcal{DG}$, $m|_{\sigma}$ satisfies the local axioms Φ^N ,
4. for each $N \xrightarrow[\text{hide}]{\sigma} M \in \mathcal{DG}$, m has a σ -expansion m' (i.e. $m'|_{\sigma} = m$) that is an N -model, and
5. for each $N \xrightarrow[\text{free}]{\sigma} M \in \mathcal{DG}$, m is an N -model that is strongly persistently $\text{Mod}(\sigma)$ -free in $\text{Mod}_{\mathcal{DG}}(N)$ ¹.

¹This notion was introduced by Dfn 4.2.1.

Complementary to definition links, which *define* the theories of related nodes, we introduce the notion of a *theorem link* with the help of which we are able to *postulate* relations between different theories. Theorem links are the central data structure to represent proof obligations arising in formal developments. Again, we distinguish between local and global theorem links (denoted by $N = \overset{\sigma}{=} \Rightarrow M$ and $N - \overset{\sigma}{=} \succ M$ respectively). We also need hiding theorem links $N = \overset{\sigma}{=} \underset{\text{hide } \vartheta}{\Rightarrow} M$ (where for some $\Sigma, \vartheta : \Sigma \rightarrow \Sigma^N$ and $\sigma : \Sigma \rightarrow \Sigma^M$). The semantics of theorem links is given by the next definition.

Definition 4.3.3 Let \mathcal{DG} be a development graph and N, M nodes in \mathcal{DG} .

- \mathcal{DG} implies a global theorem link $N = \overset{\sigma}{=} \Rightarrow M$ (denoted $\mathcal{DG} \models N = \overset{\sigma}{=} \Rightarrow M$) iff for all $m \in \mathbf{Mod}_{\mathcal{DG}}(M)$, $m|_{\sigma} \in \mathbf{Mod}_{\mathcal{DG}}(N)$;
- \mathcal{DG} implies a local theorem link $N - \overset{\sigma}{=} \succ M$ (denoted $\mathcal{DG} \models N - \overset{\sigma}{=} \succ M$) iff for all $m \in \mathbf{Mod}_{\mathcal{DG}}(M)$, $m|_{\sigma} \models \Phi^N$;
- \mathcal{DG} implies a hiding theorem link $N = \overset{\sigma}{=} \underset{\text{hide } \vartheta}{\Rightarrow} M$ (denoted $\mathcal{DG} \models N = \overset{\sigma}{=} \underset{\text{hide } \vartheta}{\Rightarrow} M$) iff for all $m \in \mathbf{Mod}_{\mathcal{DG}}(M)$, $m|_{\sigma}$ has a ϑ -expansion to a Σ^M -model, which means that there is $n \in \mathbf{Mod}_{\mathcal{DG}}(N)$ such that $N|_{\vartheta} = m|_{\sigma}$.

For our simple structuring language presented in Section 4.2, the transformation of a structured specification SP into a development graph \mathcal{DG} is defined inductively:

- if SP is a basic specification (Σ, E) , then \mathcal{DG} contains a node N for SP with $\Sigma^N = \Sigma$ and $\Phi^N = E$;
- if $SP = SP_1$ **and** SP_2 and N_1, N_2 are the nodes of SP_1 and SP_2 in \mathcal{DG} respectively, then we add a new node N with $\Sigma^N = \Sigma^{N_1} \cup \Sigma^{N_2}$ and $\Phi^N = \emptyset$ and for $i = 1, 2$, a global definition link from N_i to N labeled with $\iota_{\Sigma^{N_i}, \Sigma^N}$;
- if $SP = SP'$ **with** σ , N' is the node of SP' in \mathcal{DG} and $\sigma : \Sigma^{N'} \rightarrow \Sigma$, then we add a new node N with $\Sigma^N = \Sigma$ and $\Phi^N = \emptyset$ and a global definition link from N' to N labeled with σ ;
- if $SP = SP'$ **hide** σ , N' is the node of SP' in \mathcal{DG} and $\sigma : \Sigma \rightarrow \Sigma^{N'}$, then we add a new node N with $\Sigma^N = \Sigma$ and $\Phi^N = \emptyset$ and a hiding definition link from N' to N labeled with σ (notice that the morphism goes indeed against the direction of the link);
- if $SP =$ **free** SP' **along** σ , N' is the node of SP' in \mathcal{DG} and $\sigma : \Sigma \rightarrow \Sigma^{N'}$, then we add a new node N with $\Sigma^N = \Sigma^{N'}$ and $\Phi^N = \emptyset$ and a free definition link from N' to N labeled with σ .

Note that the semantics of development graphs is also model-theoretic. In practice, we associate to each node N in a development graph \mathcal{DG} a theory $Th_{\mathcal{DG}}(N)$

such that every model $m \in \mathbf{Mod}_{\mathcal{DG}}(N)$ is also a model of $Th_{\mathcal{DG}}(N)$. $Th_{\mathcal{DG}}(N)$ is then used for discharging the proof obligations corresponding to the node N . When the node N has no incoming hiding of free definition links, $Th_{\mathcal{DG}}(N)$ is obtained by translating the sentences of each node M in the subgraph of N to the signature Σ^N along the path from M to N . In the other case, when N has incoming hiding links, $Th_{\mathcal{DG}}(N)$ is only an approximation of the model class of N , as we discuss in Sec. 8.6. Finally, in Sec. 9.4 we present a method for normalizing free definition links in a development graph in CASL; a third case is therefore not needed here.

Example 4.3.4 *The development graph of the specifications in Example 4.2.3 is presented in Fig. 4.1. First, a node for the basic specification MONOID is introduced. GROUPINV extends MONOID, so we record this with a global definition link from the node of MONOID to the node of GROUPINV labeled with the inclusion morphism between the corresponding signatures. Finally we introduce a hiding definition link from GROUPINV to GROUP, labeled with the inclusion of the signature of GROUP to the signature of GROUPINV (so the morphism goes against the direction of the link).*



Figure 4.1: Development graph of algebraic structures example.

Development graphs are equipped with a proof calculus for discharging theorem links. Roughly, the idea is to decompose global theorem links into simpler ones by applying a number of transformation rules to the graph, until eventually theorems are introduced in the nodes and they can be proved or disproved using the tools specific to the logic of the theory of the node or by translating along institution comorphisms. The proof calculus of development graphs is proved sound and moreover complete relatively to an oracle for conservative extensions in [Mossakowski et al., 2001].

4.4 CASL Architectural Specifications

While structured specifications provide means for managing the complexity of large specifications, CASL architectural specifications [Bidoit et al., 2002a] have been introduced with the goal to prescribe the architecture of the implementations of a software system. In particular, they can also be regarded as providing means for building models in a structural way. Note that in contrast, the structure of a specification imposes no extra requirement on the structure on the models of that specification, which are treated, just like models of basic specifications, in a monolithic way.

Each architectural specification contains a number of component units together with a linking procedure which describes how to combine the components to obtain

an implementation of the overall system, thus providing *structure* for the implementation. The intuition is that architectural specifications introduce branching points in the development process and each branch can be further refined (i.e. implemented independently of development on the other branches). Each unit is given a name and assigned a specification; the intended meaning is to provide a model of the specification. Models can be parameterized, taking models as arguments and delivering models as results. Such parameterized models are *generic* units, specified by giving the a list of specifications for the arguments and a specification describing the results. The models produced by a generic unit are required to preserve the parameters (*persistency*), with the intuition that the programs passed as arguments must not be re-implemented, and the function is only defined on *compatible* models, meaning that the implementation of the parameters must be the same on common symbols. Units are combined in unit expressions with operations like renaming, hiding, amalgamation and applications of generic units. Again, terms are only defined for compatible models, in the sense that common symbols must be interpreted in the same way. Let us mention that architectural specifications are independent of the underlying formalism used for basic specifications, which is modelled as again an institution (Sec. 3.1).

$$SP \rightsquigarrow k \left\{ \begin{array}{l} U_1 : SP_1 \\ \vdots \\ U_n : SP_n \end{array} \right.$$

In the figure above, SP is the initial specification, U_1, \dots, U_n are units with their specifications SP_1, \dots, SP_n and k is the linking procedure involving the units. We record the design decision that the task of providing an implementation for SP has been split into smaller subtasks of providing models for the specifications SP_i which are then combined as prescribed by k . The refinement relation is denoted \rightsquigarrow and the refinement is correct if for any models A_i of SP_i , $i = 1, \dots, n$, the model $k(A_1, \dots, A_n)$ is indeed a model of SP . The corresponding CASL architectural specification is written:

```

arch spec ASP =
  units
     $U_1 : SP_1;$ 
    ...
     $U_n : SP_n;$ 
  result UT

```

where UT is a unit term describing the linking procedure k and possibly involving the units U_1, \dots, U_n . Note that in Chapter 5 we present an extension of CASL which allows to write not only an architectural specification, but an entire refinement step as a CASL specification.

Specifications of units consist of parameters and result specifications, given as structured specifications: if SP_1, \dots, SP_n, SP are structured specifications, then

```

ASP ::= S | units UDD1 ... UDDn result UE
UDD ::= UDEFN | UDECL
UDECL ::= UN : USP (given UT1, ..., UTn)
USP ::= SP | SP1 × ... × SPn → SP | arch spec ASP
UDEFN ::= A = UE
UE ::= UT | λ A1 : SP1, ..., An : SPn • UT
UT ::= A | A [FIT1] ... [FITn] | UT and UT | UT with σ : Σ → Σ' |
      UT hide σ : Σ → Σ' | local UDEFN1 ... UDEFNn within UT
FIT ::= UT | UT fit σ : Σ → Σ'

```

Figure 4.2: Syntax of the CASL architectural language.

$USP = SP_1 \times \dots \times SP_n \rightarrow SP$ is a unit specification. Its semantics is determined as follows:

$$\begin{array}{c}
\vdash SP_1 \triangleright \Sigma_1 \\
\vdots \\
\vdash SP_n \triangleright \Sigma_n \\
\vdash SP \triangleright \Sigma \\
\hline
\Sigma \text{ extends } \Sigma_1 \cup \dots \cup \Sigma_n \\
\vdash USP \triangleright \Sigma_1 \times \dots \times \Sigma_n \rightarrow \Sigma
\end{array}$$

where $\Sigma_1 \times \dots \times \Sigma_n \rightarrow \Sigma$, the signature of USP , is called a *unit signature* and is typically denoted by $U\Sigma$, and

$$\begin{array}{c}
\vdash USP \triangleright U\Sigma \\
\emptyset \vdash SP_1 \Rightarrow \mathcal{M}_1 \\
\vdots \\
\emptyset \vdash SP_n \Rightarrow \mathcal{M}_n \\
\mathcal{M}_0 = \mathcal{M}_1 \oplus \dots \oplus \mathcal{M}_n \\
\mathcal{M}_0 \vdash SP \Rightarrow \mathcal{M} \\
\hline
\vdash USP \Rightarrow \{F \in \text{Unit}(U\Sigma) \mid \text{for all } M \in \mathcal{M}_0, \overline{M} = \langle M|_{\Sigma_1}, \dots, M|_{\Sigma_n} \rangle \in \text{dom}(F), \\
F(\overline{M}) \in \mathcal{M} \text{ and } F(\overline{M})|_{\Sigma} = M\}
\end{array}$$

where $\text{Unit}(U\Sigma)$ is the set of all *generic units* over $U\Sigma$, i.e. partial functions taking compatible models of the argument signatures to models of the result signature in such a way that the arguments are protected. The compatibility of arguments means that the models can be amalgamated to a model of the union of all the signatures. The above definition ensures that the domain of any generic unit contains compatible tuples of models only. When $n = 0$ (no parameters), unit signatures are plain signatures and (non-generic) units are just models of the corresponding signature. We denote the model class of USP as defined in the model semantics rule by $\text{Unit}(USP)$.

Fig. 4.2 presents the complete syntax of the architectural language of CASL. Here, S stands for an architectural specification name, A for a component name, SP is a structured specification, Σ and Σ' denote signatures and σ denotes a signature

morphism. The architectural language can be shortly explained as follows: an architectural specification ASP consists of a list of unit declarations $UDECL$ with an optional list of imported units and unit definitions $UDEFN$ (where declarations assign unit specifications USP to units and definitions assign unit expressions UE to units) and a result unit expression formed with the units declared/defined. Notice that we allow the specification of a unit to be itself architectural (named or anonymous) and that for units declarations there is an optional list of imported units (marked as such by enclosing in $\langle . \rangle$). The list must be empty when USP is architectural. Unit expressions are used to give definitions for generic units, while unit terms define non-generic units. When the result unit of an architectural specification is generic, the system is “open”, requiring some parameters to provide an implementation.

A very simple example is the specification below, which prescribes the architecture of a container [Bidoit and Mosses, 2004].

```

arch spec CONTAINER =
  units
    N : NATURAL_ORDER;
    C : CONT[NATURAL_ORDER] given N
  result C

```

The task of providing an implementation for natural numbers ordered by the “less than” relation is separated from the task of providing an implementation for the containers having natural numbers as elements. Note that the component C is built assuming the existence of an implementation for the component N . The implementation of natural numbers in the container must however preserve the implementation provided by N . In particular, in the implementation of C , no other assumptions about the choices made during the implementation of N should be made, besides what is explicitly ensured by the specification $NATURAL_ORDER$.

We briefly recall the semantics of architectural specifications (see [Mosses, 2004, Bidoit et al., 2002a] for details). An architectural signature $A\Sigma$ consists of a unit signature $U\Sigma$ for the result unit together with a *static context* which is a map assigning unit signatures to the names of component units. Starting with the initial empty static context, the static semantics for declarations and definitions adds to it the signature of each new unit and the static semantics for unit terms and expressions does the type-checking in the current static context. The static semantics rules are of form $\vdash ASP \triangleright A\Sigma$ where $A\Sigma$ is an architectural signature. Model semantics is assumed to be run only after a successful run of the basic static semantics and it produces a class of architectural models \mathcal{AM} over the resulting architectural signature, where an architectural model over an architectural signature $A\Sigma$ consists of a result unit over the result unit signature and a collection of units over the signatures given in the static context, which is called a *unit environment*. The model semantics rules are of form $\vdash ASP \Rightarrow \mathcal{AM}$ with \mathcal{AM} being an architectural model. Unit terms and unit expressions denote unit evaluators UEv which build a unit over the

$$\begin{array}{c}
P_{st}(F) = \tau : \Sigma \rightarrow \Sigma' \\
C_{st} \vdash T \triangleright \Sigma^A \\
\sigma : \Sigma \rightarrow \Sigma^A \\
(\sigma_R, \tau_R, \Sigma_R) \text{ is the pushout of } (\sigma, \tau) \\
\hline
P_{st}, C_{st} \vdash F[T \text{ fit } \sigma] \triangleright \Sigma_R \\
\\
C \vdash T \Rightarrow UEv \\
\text{for each } E \in C, UEv(E)|_\sigma \in \text{dom}(E(F)) \text{ (i)} \\
\text{for each } E \in C, \text{ there is a unique } M \in \text{Mod}(\Sigma_R) \text{ such that} \\
M|_{\tau_R} = UEv(E) \text{ and } M|_{\sigma_R} = E(F)(UEv(E)|_\sigma) \text{ (ii)} \\
UEv_R = \{E \mapsto M \mid E \in C, M|_{\tau_R} = UEv(E), M|_{\sigma_R} = E(F)(UEv(E)|_\sigma)\} \\
\hline
C \vdash F[T \text{ fit } \sigma] \Rightarrow UEv_R
\end{array}$$

Figure 4.3: Basic static and model semantics rules for unit application

corresponding signature from a unit environment that records the units for the unit names that can appear in the term.

As an example, Fig. 4.3 presents the basic static semantics and model semantics rules for unit application (notice that we simplify to the case of units with just one argument). The static semantics rule produces the signature of the term T and returns as signature of $F[T]$ the pushout Σ_R of the span (σ, τ) , where τ is the unit signature of F stored in the list of parameterized unit signatures P_{st} . In the general case, the signature is obtained with the following definition.

Definition 4.4.1 Let $F : SP_1 \times \dots \times SP_n \rightarrow SP$ be a generic unit. Let $\Delta : \Sigma^F \rightarrow \Sigma$ be the inclusion of the signatures of the formal parameters into the signature of the result and at application, the fitting arguments give a signature morphism $\sigma^A : \Sigma^F \rightarrow \Sigma^A$ from the formal parameters to the actual parameters. Then, $(\Sigma(\Delta), \sigma^A, \iota_{\Sigma^A \subseteq \Sigma(\Delta)})$ form a selected pushout for (σ, Δ) :

$$\begin{array}{ccc}
& \Sigma_A & \xrightarrow{\iota_{\Sigma^A \subseteq \Sigma(\Delta)}} & \Sigma(\Delta) \\
& \uparrow \sigma & & \uparrow \sigma^A(\Delta) \\
& \Sigma_F & \xrightarrow{\Delta} & \Sigma \\
\iota_1 \nearrow & & \nwarrow \iota_n & \\
\Sigma_1 & & & \Sigma_n
\end{array}$$

In the case of the CASL logic, [Mosses, 2004] presents a set-theoretical construction of the selected pushout².

The model semantics rule first analyzes the argument T and gives a unit evaluator UEv . Then, provided that the conditions (i) the actual parameter fits the domain and (ii) the models $UEv(E)$ and $E(F)(UEv(E)|_\sigma)$ can be amalgamated to

²This works for the institutions with qualified symbols defined in [Mossakowski, 2000] too.

```

LIB-DEFN ::= lib-defn LIB-NAME LIB-ITEM*
LIB-ITEM ::= SPEC-DEFN | VIEW-DEFN | ARCH-SPEC-DEFN | UNIT-SPEC-DEFN
SPEC-DEFN ::= spec SPEC-NAME = SPEC
SPEC ::= ( $\Sigma, E$ ) | SPEC and SPEC | SPEC with  $\sigma$  | SPEC hide  $\sigma$ 
VIEW-DEFN ::= view VIEW-NAME : SPEC-NAME to SPEC-NAME =  $\sigma$ 
ARCH-SPEC-DEFN ::= arch spec ARCH-SPEC-NAME = ASP
UNIT-SPEC-DEFN ::= unit spec UNIT-SPEC-NAME = USP

```

Figure 4.4: CASL Libraries (generic specifications omitted).

a Σ_R -model M hold, the result unit evaluator UEv_R gives the amalgamation M for each unit environment $E \in C$. The condition (i) will be discharged with the help of a *proof calculus* for architectural specifications, that we present in Sec. 11.1. Typically one would expect that the conditions of type (ii) would be discarded statically. For this purpose, an *extended static semantics* was introduced in [Schröder et al., 2001], where the dependencies between units are tracked with the help of a diagram of signatures. The idea is that we can now verify that the interpretation of two symbols is the same by looking for a “common origin” in the diagram, i.e. a symbol which is mapped via some paths to both of them.

Note that the assumption that generic units are interpreted as functions requires that a generic unit yields the same result when applied to the same arguments. However, the extended static semantics is sound and complete only w.r.t. a *generative semantics*, where applying a generic unit to the same arguments provides possible different results. If all generic units are applied only once, the generative and the applicative (non-generic) semantics are equivalent. We denote $\vdash ASP \Rightarrow_g \mathcal{AM}$ the generative model semantics of architectural specifications.

For any architectural specification ASP , we denote $|ASP|$ the specification obtained by removing everything but the signature from the specifications used in declarations. Generic units can be interpreted as total functions by introducing an additional value \perp - this ensures consistency of generic unit specifications in $|ASP|$ whenever the unit specification is already consistent in an architectural specification ASP and is called *partial model semantics* in [Mosses, 2004], Section IV:5.

Note that both structured and architectural specifications are named. This is managed by organizing specifications in *libraries* of specifications. A library consists of a list of definitions of named specifications. The names of the specifications must be distinct and the visibility is linear. Fig. 4.4 gives the grammar of library definitions.

4.5 HETCASL

A key feature of CASL is that syntax and semantics of its language constructs are formulated over an arbitrary institution. Intuitively, HETCASL can be regarded as an extension of the variant of CASL obtained by replacing at the basic level the

institution SubPCFOL_{ms} with the Grothendieck institution over a graph of logic and their translations.

We assume that the logic graph contains the institution SubPCFOL_{ms} and it marks it as *default logic* and that some of the logic translations in the logic graph are marked as *default logic inclusions*, with the extra requirement that at most one translation between two logics can be marked as default inclusion. Default logic inclusions are used to *coerce* a specification by translating it along the corresponding logic inclusion. We also assume a partial union operation on the logics of the logic graph such that if the union of two logics is defined, both logics are included in their union and this union is minimal with this property.

The syntax of heterogeneous specifications can be explained as follows. At the library level, we add the possibility that the user selects the current logic (`logic ID`, where `ID` stands for a logic name), which is then used for parsing and analyzing the subsequent definitions. If no current logic is specified, the default logic is assumed. A translation along a logic comorphism is written `SPEC with logic ID`. In this case, `ID` stands (1) for a comorphism name, or (2) for an anonymous comorphism given just by specifying the source and target logics (`L1 -> L2`), which selects the unique translation between the two logics, or (3) for the default inclusion comorphism from the current logic to the specified one (`->L1`). The semantics of `SPEC with logic ID` has been already introduced in Def. 4.2.4, where ρ is now the comorphism identified by `ID` as explained above. Then the syntax for Grothendieck signature morphisms combines an inter-logic translation, written using `SPEC with logic ID`, with an intra-logic translation, where the (now homogeneous) signature morphism is written in the usual CASL syntax (see [Mosses, 2004]).

Heterogeneous proving is done in the Grothendieck institution also using the formalism of development graphs. Conditions needed for completeness of development graph calculus for the heterogeneous case have been investigated in [Mossakowski, 2002a, 2005].

It should be stressed that the name “HETCASL” only refers to CASL’s structuring constructs. The individual logics used in connection with HETCASL and HETS can be completely orthogonal to CASL.

C_{ASL} Refinement Language

Contents

5.1 Syntax	61
5.2 Semantics	68

In this chapter we present the refinement language for C_{ASL} introduced in [Mossakowski et al., 2005]. The language can be regarded as an extension of the architectural language, which it also subsumes. As the refinement language plays a central role in this thesis, we decided to recall its static and model semantics in full detail, again following the cited paper.

The issue of refinement has been discussed in many specification frameworks, starting with [Hoare, 1972] and [Milner, 1971], and some frameworks provide methods for proving correctness of refinements. But this is normally regarded as a “meta-level” issue and specification languages have typically not included syntactic constructs for formally stating such relationships between specifications that are analogous to those presented here for C_{ASL}. A notable exception is Specware [Smith, 1999], where specifications (and implementations) are structured using specification diagrams, and refinements correspond to specification morphisms for which syntax is provided. This, together with features for expanding specification diagrams, provides sufficient expressive power to capture our branching specification refinements. A difference is that Specware does not include a distinction between structured specifications and C_{ASL}-like architectural specifications, and refinements are required to preserve specification structure.

5.1 Syntax

The aim of the refinement language is to formalize development tress. As we discussed in Sec. 1.3, the basic idea is to start with a specification of requirements and to add more details in a stepwise manner until we reach a specification concrete enough to be (relatively) easily translated into a program. The degree of abstraction thus decreases at each step and as consequence, the class of models narrows with each design decision. For this reason, we may refer to the specification being refined as being more “abstract” than the specification we refine to. We can record such refinements already using views, described in Sec. 4.2. Moreover, as we have seen in Sec. 1.3, it is practically sensible to split up an implementation in a number

of components, to be further developed individually. Towards this purpose, CASL already provides architectural specifications, described in Sec. 4.4.

However, this does not suffice for all purposes. Firstly, views formalize only refinement between non-generic unit specifications. Secondly, there is no way to record that a structured specification is refined to an architectural specification. Finally, architectural specifications only specify individual branching points. While we can specify that a component of an architectural specification is further decomposed into sub-components by making its own specification architectural, the drawback is that this decision must be specified within a single architectural specification, that captures the whole development process.

In contrast, the CASL refinement language provides more flexibility. Refinement of generic unit specifications and refinement to an architectural specification are primitives of the language. *Compositions* of refinements are also made explicit, using several alternative constructions. The language also permits an “a posteriori” refinement of components, that is, we can further refine the unit specifications appearing in an architectural specification *after* we have written the architectural specification and without having to re-write it.

Example 5.1.1 *We will illustrate the CASL architectural and refinement languages with the help of an industrial case study: specification of a steam boiler control system for controlling the water level in a steam boiler. The problem has been formulated in [Abrial et al., 1996] as a benchmark for specification languages; [Bidoit and Mosses, 2004] give a complete solution using CASL, including architectural design and refinement of components. However, the refinement steps were presented only in an informal way.*

The specifications involved can be briefly explained as follows. VALUE specifies in a very abstract way a sort Value and some operations and predicates on values. This specification acts as a parameter of the entire design. PRELIMINARY gathers the messages in the system, both sent and received, and also defines a series of constants characterizing the steam boiler. SBCS_STATE introduces observers for the system states, while SBCS_ANALYSIS extends this to an analysis of the messages received, failure detection and computation of messages to be sent. Finally, STEAM_BOILER_CONTROL_SYSTEM specifies the initial state and the reachability relation between states. We record the requirement that the system is open in the models for VALUE using a generic unit specification ¹:

unit spec SBCS_OPEN = VALUE → STEAM_BOILER_CONTROL_SYSTEM

The initial design for the architecture of the system is recorded by the following architectural specification:

arch spec ARCH_SBCS =
units P : VALUE → PRELIMINARY;

¹The complete specification of the SBCS example can be found in the Appendix B.

```

S : PRELIMINARY → SBCS_STATE;
A : SBCS_STATE → SBCS_ANALYSIS;
C : SBCS_ANALYSIS → STEAM_BOILER_CONTROL_SYSTEM
result λ V : VALUE • C [A [S [P [V]]]]

```

Here, the units P, S, A and C are all generic units. Moreover, the components are combined in the way prescribed in the result unit of ARCH_SBCS; a model of VALUE is required to able to provide a model of the entire system.

The first kind of refinement introduced by the CASL refinement language is *unit specification refinement*. This is written *USP refined via σ to USP'*, where *USP* and *USP'* are unit types of form $SP_1 \times \dots \times SP_n \rightarrow SP$ and $SP_1 \times \dots \times SP_n \rightarrow SP'$ respectively and $\sigma : Sig(SP) \rightarrow Sig(SP')$. When σ is missing, it is assumed to be the identity. This also covers the case of non-generic unit types by taking $n = 0$. We make the simplifying assumption that the parameter specifications of generic unit specifications do not change under refinement. This allows us to freely use the reduct notation $U|_\sigma$, for generic units $U \in Unit(USP)$ as well; in this case, the notation denotes the unit function obtained by reducing the result via σ after applying U . In practice, this restriction is not troublesome, since we always can write an architectural specification that adjusts the parameter specification as required, as we will show below. We can express then correctness of a unit specification refinement as

$$U|_\sigma \in Unit(USP) \text{ for each unit } U \in Unit(USP')$$

and denote this by $USP \rightsquigarrow_\sigma USP'$. If *USP* and *USP'* are non-generic, we can express the refinement *USP refined via σ to USP'* equivalently as **view $V : USP$ to $USP' = \sigma$** . Therefore, simple refinements can be regarded as a generalization of views to the generic case.

Two refinements can be combined to form a *chain* of refinements, which captures the situation $USP \rightsquigarrow_\sigma USP' \rightsquigarrow_\tau USP''$. We write this *USP refined via σ to USP' refined via τ to USP''*, or using *named* refinements as one of the three equivalent constructions C1, C2 or C3 below:

```

refinement R1 = USP refined via  $\sigma$  to USP'
refinement R2 = USP' refined via  $\tau$  to USP''
refinement C1 = USP refined via  $\sigma$  to R2
refinement C2 = USP refined via  $\sigma$  to USP' then R2
refinement C3 = R1 then R2

```

where the last two alternatives make use of an operation of *composition* of refinement specifications, using the keyword **then**. The semantics rules, which will be presented in Sec. 5.2, enforce which kind of compositions are legal.

As a next step, we can introduce branching in development by refining to an architectural specification:

```

refinement R = USP refined via  $\sigma$  to arch spec ASP

```

The architectural language already permits that the specification of a component unit of *ASP* is itself architectural. This means that we are allowed to record in an architectural specification decisions regarding the design of a component. Since we have a more expressive language at hand for such design decisions, it is natural to generalize this by allowing the specifications of the component units of *ASP* to be refinements themselves:

refinement $R = USP$ refined via σ to USP'

arch spec $ASP = \{$
 units $U : R$
 ... $\}$

These two constructions give us the second kind of refinements, *branching specification refinements*.

Finally, we also allow to further refine the (named) components of architectural specifications using *component refinements*, which are written $\{UN_i \text{ to } SPR_i\}_{i \in \mathcal{J}}$, where UN_i stands for a unit name and SPR_i for a refinement. This expresses that for each $i \in \mathcal{J}$, the component UN_i of the architectural specification at hand is further refined to SPR_i . This is usually recorded using a composition of the form

refinement $R = \text{arch spec } ASP \text{ then } \{UN_i \text{ to } SPR_i\}$

which specifies that the component UN_i of *ASP* is refined to SPR_i . The semantics rules prevent the specification of a unit in an architectural specification to be a component refinement, that is, only simple or branching refinements are allowed.

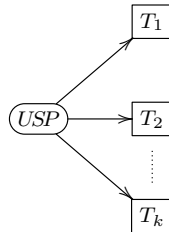
All this can be intuitively summed up as follows. The refinement language for CASL provide means for specifying development trees which are constructed using three types of building blocks:

- simple refinement steps, which can be graphically represented as



where $USP \rightsquigarrow^{\sigma} USP'$ and the double arrow denotes a refinement,

- branching refinements, which can be represented as



where T_i are themselves development trees and the simple arrow denotes an architectural decomposition

- component refinements, which are families of named development trees and can be represented as

$$C = \begin{cases} UN_1 \mapsto C_1 \\ UN_2 \mapsto C_2 \\ \dots \\ UN_k \mapsto C_k = \begin{cases} UN_k^1 \mapsto \boxed{T_1} \\ \dots \\ UN_k^n \mapsto C_k^n \end{cases} \end{cases}$$

where UN_i are unit names and C_i are either development trees or themselves families of trees (as we made explicit in the case of C_k);

which can be combined using composition of refinements, corresponding to putting together subtrees that match at the connection points. We will make all this formal in Sect. 11.4.

Example 5.1.2 (Refinement of arbitrary unit types) Given two unit specifications $SP \rightarrow SP'$ and $SP_1 \rightarrow SP'_1$ with a specification morphism $\sigma : SP_1 \rightarrow SP$, the following is a correct specification refinement²:

unit spec $USP = SP \rightarrow SP'$

unit spec $USP' = SP_1 \rightarrow SP'_1$

refinement $R = USP$ refined via τ to arch spec{

units $F : USP'$

result lambda $X : SP \bullet F [X \text{ fit } \sigma]$ }

where τ is a specification morphism from SP_1 to the pushout specification $SP'_1 \oplus SP$ in the following diagram:

$$\begin{array}{ccc} SP_1 & \longrightarrow & SP'_1 \\ \downarrow \sigma & & \downarrow \text{dotted} \\ SP & \dashrightarrow & SP'_1 \oplus SP \\ & \searrow & \uparrow \tau \\ & & SP' \end{array}$$

The complete syntax for CASL refinements is presented in Fig. 5.1, where A, A_1, A_n stand for unit names and σ for a signature morphism. We eliminate architectural specifications as an alternative of unit specifications, since they are a particular case of refinements. This is also why we can now speak of an architectural refinement level subsuming the architectural level in the CASL language. Moreover, we consider units with imports in this chapter only a syntactic construction. Their semantics will be introduced in detail in Sec. 10.5.

²Assuming that all symbols shared between SP'_1 and SP originate in SP_1 , as imposed by CASL rules for application of generic units.

```

LIB-ITEM ::= ... | REF-SPEC-DEFN
REF-SPEC-DEFN ::= refinement REF-SPEC-NAME = SPR
SPR ::= USP | arch spec ASP | SPR then SPR |
        USP refined ⟨via  $\sigma$ ⟩ to SPR | { $A_1$  to SPR1, ...,  $A_n$  to SPRn}
UDECL ::=  $A$  : SPR |  $A$  : USP given UT1, ..., UTn
USP ::= SP | SP1 × ... × SPn → SP | arch spec ASP

```

Figure 5.1: Syntax of the CASL refinement language.

Example 5.1.3 We illustrate the CASL refinement language with the help of an example from [Mossakowski et al., 2005]. We start with a loose specification of monoids:

```

spec MONOID =
  sort Elem
  ops 0 : Elem;
        __+__ : Elem × Elem → Elem, assoc, unit 0

```

then natural numbers are specified with the usual Peano axioms, addition is defined, and finally the successor is hidden:

```

spec NATWITHSUC =
  free type Nat ::= 0 | suc(Nat) %% shorthand for Peano axioms
  op __+__ : Nat × Nat → Nat, unit 0
       $\forall x, y : \text{Nat} \bullet x + \text{suc}(y) = \text{suc}(x + y)$ 
spec NAT = NATWITHSUC hide suc

```

and we can record that natural numbers with addition form a monoid as follows:

```

refinement R1 = MONOID refined via Elem ↦ Nat to NAT

```

Natural numbers are further implemented as lists of binary digits, constructed by two postfix operations `__0` and `__1`. `__+__` is addition, `__++__` is addition with carry bit.

```

spec NATBIN =
  generated type Bin ::= 0 | 1 | __0(Bin) | __1(Bin)
  ops __+__, __++__ : Bin × Bin → Bin
   $\forall x, y : \text{Bin}$ 
  • 0 0 = 0
  • 0 1 = 1
  •  $\neg 0 = 1$ 
  •  $x 0 = y 0 \Rightarrow x = y$ 
  •  $\neg x 0 = y 1$ 
  •  $x 1 = y 1 \Rightarrow x = y$ 
  • 0 + 0 = 0
  • 0 ++ 0 = 1

```

- $x\ 0 + y\ 0 = (x + y)\ 0$
- $x\ 0 ++ y\ 0 = (x + y)\ 1$
- $x\ 0 + y\ 1 = (x + y)\ 1$
- $x\ 0 ++ y\ 1 = (x ++ y)\ 0$
- $x\ 1 + y\ 0 = (x + y)\ 1$
- $x\ 1 ++ y\ 0 = (x ++ y)\ 0$
- $x\ 1 + y\ 1 = (x ++ y)\ 0$
- $x\ 1 ++ y\ 1 = (x ++ y)\ 1$

and we obtain thus a new refinement:

refinement R2 = NAT **refined via** Nat \mapsto Bin **to** NATBIN

which can be composed with the first one to form a chain:

refinement R3 = R1 **then** R2

Furthermore, we can require that the addition should be implemented first, and then the successor defined in terms of addition:

arch spec ADDITION_FIRST =
units N : NAT;
 F : NAT \rightarrow NATWITHSUC
result F[N]

and we can record this design decision as:

refinement R4 = NATWITHSUC **then arch spec** ADDITION_FIRST

We can then further refine the component N of ADDITION_FIRST:

refinement RCOMP = {N **to** R2}
refinement R5 = R4 **then** RCOMP

Example 5.1.4 We write the initial refinement of the steam boiler system as

refinement REF_SBCS = SBCS_OPEN **refined to arch spec** ARCH_SBCS

We proceed with refining the individual units. The specifications of C and S in ARCH_SBCS above do not require further architectural decomposition. The specification of S, recorded in the unit specification STATE_ABSTR, can be refined by providing an implementation of states as a record of all observable values. This is done in SBCS_STATE_IMPL, assuming an implementation of PRELIMINARY; we record this development in the unit specification UNIT_SBCS_STATE. The refinement of S is then written in STATE_REF:

unit spec STATE_ABSTR = PRELIMINARY \rightarrow SBCS_STATE
unit spec UNIT_SBCS_STATE =
 PRELIMINARY \rightarrow SBCS_STATE_IMPL
refinement STATE_REF =
 STATE_ABSTR **refined to** UNIT_SBCS_STATE

For the units P and A , we proceed with designing their architecture. This is recorded in the architectural specifications $ARCH_ANALYSIS$:

```
arch spec ARCH_ANALYSIS =
  units FD : SBCS_STATE → FAILURE_DETECTION;
  PR : FAILURE_DETECTION → PU_PREDICTION;
  ME : PU_PREDICTION → MODE_EVOLUTION[PU_PREDICTION];
  MTS : MODE_EVOLUTION[PU_PREDICTION] → SBCS_ANALYSIS
  result λ S : SBCS_STATE • MTS [ME [PR [FD [S]]]]
```

and $ARCH_PRELIMINARY$:

```
arch spec ARCH_PRELIMINARY =
  units SET : {sort Elem} × NAT → SET[sort Elem];
  B : BASICS;
  MS : MESSAGES_SENT given B;
  MR : VALUE → MESSAGES_RECEIVED given B;
  CST : VALUE → SBCS_CONSTANTS
  result λ V : VALUE
  • SET [MS fit Elem ↦ S_Message] [V]
  and SET [MR [V] fit Elem ↦ R_Message] [V]
  and CST [V]
```

We can now record the component refinement:

```
refinement REF_SBCS' = REF_SBCS then
  {P to arch spec ARCH_PRELIMINARY, S to STATEREF,
  A to arch spec ARCH_ANALYSIS}
```

Moreover, the components FD and PR of $ARCH_ANALYSIS$ are further refined (the architectural specifications $ARCH_FAILURE_DETECTION$ and $ARCH_ANALYSIS$ are omitted, but their units are visible in Fig. 11.5):

```
refinement REF_SBCS'' =
  REF_SBCS'
  then {A to
    {FD to arch spec ARCH_FAILURE_DETECTION,
    PR to arch spec ARCH_PREDICTION }}
```

Note that this is a corrected version of an example in [Mossakowski et al., 2005].

5.2 Semantics

We now recall the semantics of refinement specifications as defined in [Mossakowski et al., 2005]. Both the *refinement signatures*, that will be used for

the static semantics, and the *refinement relations*, which provide model semantics of refinements, come in three variants, to correspond with the three types of refinements.

Let us recall that refinement specification formalize development trees. The intuition behind refinement signatures is that they store the signatures of the roots and the leaves of these trees, which have to match when the trees are combined.

Definition 5.2.1 [Mossakowski et al., 2005] A refinement signature can have one of the following forms:

$$\begin{aligned} R\Sigma &::= (U\Sigma, B\Sigma) \mid \{UN_i \mapsto R\Sigma_i\}_{i \in \mathcal{J}} \\ B\Sigma &::= U\Sigma \mid \{UN_i \mapsto B\Sigma_i\}_{i \in \mathcal{J}} \end{aligned}$$

This means that a refinement signature $R\Sigma$ can be either

- a *branching refinement signature* $(U\Sigma, B\Sigma)$ where on the second component we have a *branching signature* $B\Sigma$ which can itself be either (i) a unit signature $U\Sigma'$, and in this case $R\Sigma$ is called a *unit refinement signature* or (ii) a map $\{UN_i \mapsto B\Sigma_i\}_{i \in \mathcal{J}}$ called *branching static context* and denoted $BstC$, assigning branching signatures to unit names. The intuition is that a unit refinement signature stores the signatures of a unit before and after refinement, and a branching signature generalizes this to the case when a branching is introduced by storing the signatures of all components in a map where they can be retrieved by the corresponding component's name;
- a *component refinement signature* $\{UN_i \mapsto R\Sigma_i\}_{i \in \mathcal{J}}$, storing the refinement signature of each component to be refined. When all $R\Sigma_i, i \in \mathcal{J}$ are branching refinement signatures $(U\Sigma_i, B\Sigma_i)$, we refer to the component refinement signature $\{UN_i \mapsto R\Sigma_i\}_{i \in \mathcal{J}}$ as a *refined-unit static context*, denoted $RstC$, which can then be naturally coerced to a static context $\pi_1(RstC) = \{UN_i \mapsto U\Sigma_i\}_{i \in \mathcal{J}}$ as well to a branching static context $\pi_2(RstC) = \{UN_i \mapsto B\Sigma_i\}_{i \in \mathcal{J}}$.

The rules for static semantics of refinements are then presented in Fig. 5.2. The result of the analysis is a refinement signature $R\Sigma$ and the general format of the rules is $\vdash SPR \triangleright R\Sigma$. When we only want to check that the rules apply successfully and the result is not important, we will denote this by $\vdash SPR \triangleright \square$.

We have a rule for each alternative of refinement and the rules are defined inductively on the structure of the refinement. The rule for architectural specifications hides a number of necessary changes to the semantics of architectural specifications, as given in [Mosses, 2004], Sec. III.5. This is because now the specifications of the units are either simple or branching refinements, and we get thus a refined-unit static context $RstC$ for the units of an architectural specification, as we illustrate here with the rule for the static semantics of unit declarations:

$$\frac{\Gamma_s \vdash SPR \triangleright R\Sigma \quad UN \notin Dom(RstC)}{\Gamma_s, RstC \vdash UN : SPR \triangleright \{UN \mapsto R\Sigma\}}$$

$$\begin{array}{c}
\frac{\vdash USP \triangleright U\Sigma}{\vdash USP \text{ qua } SPR \triangleright (U\Sigma, U\Sigma)} \quad \frac{\begin{array}{c} \vdash USP \triangleright (U\Sigma, U\Sigma) \\ U\Sigma = (\Sigma_1, \dots, \Sigma_n \rightarrow \Sigma) \\ \sigma : \Sigma \rightarrow \Sigma' \\ \vdash SPR \triangleright (U\Sigma', B\Sigma') \\ U\Sigma' = (\Sigma_1, \dots, \Sigma_n \rightarrow \Sigma') \end{array}}{\vdash USP \text{ refined via } \sigma \text{ to } SPR \triangleright (U\Sigma, B\Sigma'')} \\
\\
\frac{\vdash ASP \triangleright (U\Sigma, RstC)}{\vdash \text{arch spec } ASP \triangleright (U\Sigma, \pi_2(RstC))} \quad \frac{\begin{array}{c} \vdash SPR_1 \triangleright R\Sigma_1 \\ \vdash SPR_2 \triangleright R\Sigma_2 \\ R\Sigma = R\Sigma_1 ; R\Sigma_2 \end{array}}{\vdash SPR_1 \text{ then } SPR_2 \triangleright R\Sigma} \\
\\
\frac{\begin{array}{c} UN_1, \dots, UN_n \text{ are distinct} \\ \vdash SPR_i \triangleright R\Sigma_i, i = 1, \dots, k \end{array}}{\vdash \{UN_1 \text{ to } SPR_1, \dots, UN_k \text{ to } SPR_k\} \triangleright \{UN_1 \mapsto R\Sigma_1, \dots, UN_k \mapsto R\Sigma_k\}}
\end{array}$$

Figure 5.2: Static semantics of CASL refinements

We can then coerce $RstC$ to a static context using the projection to the first component π_1 , as below:

$$\frac{RstC(UN) = R\Sigma}{\Gamma_s, RstC \vdash UN \triangleright \pi_1(R\Sigma)}$$

For unit definitions, we do not want to allow to further refine the defined unit, and we use the \perp sign to mark that:

$$\frac{\begin{array}{c} \Gamma_s, RstC \vdash UE \triangleright \Sigma \\ UN \notin Dom(RstC) \end{array}}{\Gamma_s, RstC \vdash UN = UE \triangleright \{UN \mapsto (\Sigma, \perp)\}}$$

This ensures that the composition of a branching refinement signature with a component refinement signature having the name of a defined unit in its domain is illegal. The other static semantics rules for architectural specifications can be modified in a similar way; since this is rather straightforward, using the projection π_1 , we will not present this in detail here. The signature of the result unit expression of the architectural specification is then paired with the projection on the second component of the refined-unit static context to obtain a branching signature which is then given as result of the analysis. Finally, the complexity of the rule for refinement composition is hidden in an auxiliary partial composition operation on refinement signatures, that can be explained intuitively as checking that, at each connection point, the corresponding unit signatures match.

Definition 5.2.2 [Mossakowski et al., 2005] Given refinement signatures $R\Sigma_1$ and $R\Sigma_2$, their composition $R\Sigma_1 ; R\Sigma_2$ is defined inductively on the form of the first argument.

- $R\Sigma_1 = (U\Sigma, U\Sigma')$: then $R\Sigma_1 ; R\Sigma_2$ is defined only if $R\Sigma_2$ is a branching refinement signature of the form $(U\Sigma', B\Sigma'')$. Then $R\Sigma_1 ; R\Sigma_2 = (U\Sigma, B\Sigma'')$.
- $R\Sigma_1 = (U\Sigma, BstC')$: then $R\Sigma_1 ; R\Sigma_2$ is defined only if $R\Sigma_2$ is a component refinement signature such that $R\Sigma_2$ matches $BstC'$, i.e., $dom(R\Sigma_2) \subseteq dom(BstC')$ and for each $UN \in dom(R\Sigma_2)$,
 - either $BstC'(UN)$ is a unit signature and then $R\Sigma_2(UN) = (U\Sigma', B\Sigma'')$ with $U\Sigma' = BstC'(UN)$, or
 - $BstC'(UN)$ is a branching static context and then $R\Sigma_2(UN)$ matches $BstC'(UN)$,

Then $R\Sigma_1 ; R\Sigma_2 = (U\Sigma, BstC'[R\Sigma_2])$, where given any branching static context $BstC'$ and component refinement signature $R\Sigma_2$ that matches $BstC'$, $BstC'[R\Sigma_2]$ modifies $BstC'$ on each $UN \in dom(R\Sigma_2)$ as follows:

- if $BstC'(UN)$ is a unit signature then $BstC'[R\Sigma_2](UN) = B\Sigma''$ where $R\Sigma_2(UN) = (BstC'(UN), B\Sigma'')$,
- if $BstC'(UN)$ is a branching static context then $BstC'[R\Sigma_2](UN) = BstC'(UN)[R\Sigma_2(UN)]$.

Finally, since defined units should not be further refined, we make illegal the composition of a branching refinement specification with a component refinement specification if the latter has the name of a defined unit in its domain.

- $R\Sigma_1$ is a component refinement signature: then $R\Sigma_1 ; R\Sigma_2$ is defined only if $R\Sigma_2$ is a component refinement signature too, and moreover, for all $UN \in dom(R\Sigma_1) \cap dom(R\Sigma_2)$, $R\Sigma_{UN} = R\Sigma_1(UN) ; R\Sigma_2(UN)$ is defined. Then $R\Sigma_1 ; R\Sigma_2$ modifies the (ill-defined) union of $R\Sigma_1$ and $R\Sigma_2$ by putting $(R\Sigma_1 ; R\Sigma_2)(UN) = R\Sigma_{UN}$ for $UN \in dom(R\Sigma_1) \cap dom(R\Sigma_2)$.

Example 5.2.3 Let us apply the static semantic rules for the refinements in Ex. 5.1.3. For each of the specifications involved, we denote its unit signature with the name of the specifications, that is $\vdash \text{NAT} \triangleright \text{NAT}$ and so on. Then

- With the rule for unit specifications as refinement specifications we obtain $\vdash \text{MONOID} \text{ qua } \text{SPEC-REF} \triangleright (\text{MONOID}, \text{MONOID})$ and $\vdash \text{NAT} \text{ qua } \text{SPEC-REF} \triangleright (\text{NAT}, \text{NAT})$. Then with the rule for simple refinements we get that $\vdash \text{R1} \triangleright (\text{MONOID}, \text{NAT})$;
- With a similar reasoning, we get $\vdash \text{R2} \triangleright (\text{NAT}, \text{NATBIN})$;

- Since R3 is a composition of refinements, we have to compose the signatures of R1 and R2. The first case in the definition of composition of refinement signatures applies and since the target signature of R1 and the source signature of R2 are both NAT, the composition is legal and we get $\vdash R3 \triangleright (\text{MONOID}, \text{NATBIN})$;
- For ADDITION_FIRST, the refinement specifications of the units are analyzed first. Since they are both unit specifications, we get a refined-unit static context $RstC$ which assigns (NAT, NAT) to N and $(\text{NAT} \rightarrow \text{NATWITHSUC}, \text{NAT} \rightarrow \text{NATWITHSUC})$ to F . According to the rule for architectural specifications, this refined-unit static context is projected on the second component, thus resulting a branching static context $BstC$ which assigns NAT to N and $\text{NAT} \rightarrow \text{NATWITHSUC}$ to F . We get thus $\vdash \text{ADDITION_FIRST} \triangleright (\text{NATWITHSUC}, \{N \mapsto \text{NAT}, F \mapsto \text{NAT} \rightarrow \text{NATWITHSUC}\})$. This is also the signature of R4, using the rule for refinements;
- We have that $\vdash R2 \triangleright (\text{NAT}, \text{NATBIN})$, so according to the rule for component refinement specifications we get $\vdash \text{RCOMP} \triangleright \{N \mapsto (\text{NAT}, \text{NATBIN})\}$;
- Finally, we have to make the composition of the signature of R4 with the signature of RCOMP. We are in the second case of the definition of the composition of refinement signatures and the name N is indeed in the domain of the branching static context $BstC$ which is on the second position in the signature of R4. Moreover, $BstC(N) = \text{NAT}$ and thus matches the source signature of RCOMP. Then by definition we update the signature of N in $BstC$ to NATBIN and get $\vdash R5 \triangleright (\text{NATWITHSUC}, \{N \mapsto \text{NATBIN}, F \mapsto \text{NAT} \rightarrow \text{NATWITHSUC}\})$.

For the model semantics, we first introduce the notion of *constructor implementation* [Sannella and Tarlecki, 1988a, 2012]. Constructors are simply partial functions taking models to models, of form $\kappa : \mathbf{Mod}(\Sigma) \rightarrow \mathbf{Mod}(\Sigma')$. An example of such a constructor is already provided by the model reduct along a signature morphism $\sigma : \Sigma \rightarrow \Sigma'$, which takes any Σ' -model M' to its Σ -reduct $M'|_\sigma$.

Definition 5.2.4 [Sannella and Tarlecki, 1988a] *Let SP, SP' be specifications such that $\vdash SP \triangleright \Sigma$ and $\vdash SP' \triangleright \Sigma'$ and let $\kappa : \mathbf{Mod}(\Sigma') \rightarrow \mathbf{Mod}(\Sigma)$ be a constructor. We say that SP' is a constructor implementation of SP along κ , denoted $SP \rightsquigarrow_\kappa SP'$, if for any $M \in \mathbf{Mod}(SP')$, $M \in \text{dom}(\kappa)$ and $\kappa(M) \in \mathbf{Mod}(SP)$.*

Note that by definition we have that $\sigma : SP \rightarrow SP'$ is a morphism of specifications iff $SP \rightsquigarrow_\kappa SP'$ with $\kappa = _ |_\sigma$. These kind of constructors are specified using unit specification refinements.

For capturing branchings as well, we need n -ary constructors of form $\kappa : \mathbf{Mod}(\Sigma_1) \times \dots \times \mathbf{Mod}(\Sigma_n) \rightarrow \mathbf{Mod}(\Sigma)$, which are partial functions mapping *compatible* models to Σ -models, where the compatibility requirement means that the arguments can be amalgamated to a model of the union of signatures $\Sigma_1, \dots, \Sigma_n$. An implementation is now correct if given Σ_i -specifications SP_i , for $i = 1, \dots, n$,

any compatible models M_1, \dots, M_n of SP_1, \dots, SP_n respectively are in the domain of κ and $\kappa(M_1, \dots, M_n)$ is a model of the specification SP which we want to refine to SP_1, \dots, SP_n . These constructors are specified using branching specification refinements.

Constructors provide then an intuitive notion of *refinement model*. We can observe this best in the case of unary constructors: if $SP \rightsquigarrow_{\kappa} SP'$, we denote \mathcal{G} the graph of κ restricted to $\mathbf{Mod}(SP')$, that is, pairs of form $(M, \kappa(M))$, where $M \in \mathbf{Mod}(SP')$. We then take the inverse relation \mathcal{G}^{-1} to obtain on the first component models over $\mathbf{Sig}(SP)$ and on the second component models over $\mathbf{Sig}(SP')$, matching thus the order of models with the order of their unit signatures in the corresponding refinement signature. This generalizes to n -ary constructors to cover branching refinement and to families of constructors, which will be models of component refinement specifications. This leads us to the following formal definition.

Definition 5.2.5 [Mossakowski et al., 2005] *Given a refinement signature $R\Sigma$, we define refinement relations, \mathcal{R} , as classes of assignments, R , which take the following forms:*

- branching assignments, for $R\Sigma = (U\Sigma, B\Sigma')$, are pairs (U, BM') , where U is a unit over the unit signature $U\Sigma$ and BM' is a branching model over the branching signature $B\Sigma'$, which is either a unit over $B\Sigma'$ when $B\Sigma'$ is a unit signature (in which case the branching assignment is a unit assignment), or a branching environment BE' that fits $B\Sigma'$ when $B\Sigma'$ is a branching static context. Branching environments are (finite) maps assigning branching models to unit names, with the obvious requirements to ensure compatibility with the branching signatures indicated in the corresponding branching static context. Moreover, we require that whenever (u, bm) and (u', bm) are in \mathcal{R} we have that $u = u'$.
- component assignments, for $R\Sigma = \{UN_i \mapsto R\Sigma_i\}_{i \in \mathcal{J}}$, are (finite) maps $\{UN_i \mapsto R_i\}_{i \in \mathcal{J}}$ from unit names to assignments over the respective refinement signatures. When $R\Sigma$ is a refined-unit static context (and so each $R_i, i \in \mathcal{J}$, is a branching assignment) we refer to $RE = \{UN_i \mapsto (U_i, BM_i)\}_{i \in \mathcal{J}}$ as a refined-unit environment. Any such refined-unit environment can be naturally coerced to a unit environment $\pi_1(RE) = \{UN_i \mapsto U_i\}_{i \in \mathcal{J}}$ of the plain CASL semantics, as well as to a branching environment $\pi_2(RE) = \{UN_i \mapsto BM_i\}_{i \in \mathcal{J}}$.

The model semantics rules for refinements are presented in Fig. 5.3. They are again defined inductively on the structure of the refinements and the judgements are now of the form $\vdash SPR \Rightarrow \mathcal{R}$, where SPR is a refinement and \mathcal{R} is a refinement relation. Again, we need to modify the model semantics rules for architectural specifications, as given in [Mosses, 2004], Sec. III.5. We build a refined-unit environment RE for the units of an architectural specification that we coerce to a unit environment using the projection to the first component π_1 . The model semantics rule for unit declarations produces the context that consists of all refined-unit

$$\begin{array}{c}
\frac{\vdash USP \Rightarrow \mathcal{U}}{\vdash USP \text{ qua } SPR \Rightarrow \{(U, U) \mid U \in \mathcal{U}\}} \\
\\
\frac{\vdash USP \Rightarrow \mathcal{U} \quad \sigma : \Sigma \rightarrow \Sigma' \quad \vdash SPR \Rightarrow \mathcal{R} \\
\quad U' |_{\sigma} \in \mathcal{U}, \text{ for all } (U', BM'') \in \mathcal{R} \\
\quad \mathcal{R}' = \{(U' |_{\sigma}, BM'') \mid (U', BM'') \in \mathcal{R}\}}{\vdash USP \text{ refined via } \sigma \text{ to } SPR \Rightarrow \mathcal{R}'} \\
\\
\frac{\vdash ASP \Rightarrow \mathcal{AM}}{\vdash \text{arch spec } ASP \Rightarrow \{(U, \pi_2(RE)) \mid (U, RE) \in \mathcal{AM}\}} \\
\\
\frac{\vdash SPR_1 \Rightarrow \mathcal{R}_1 \quad \dots \quad \vdash SPR_n \Rightarrow \mathcal{R}_n}{\vdash \{UN_1 \text{ to } SPR_1, \dots, UN_n \text{ to } SPR_n\} \Rightarrow \{R \mid \text{dom}(R) = \{UN_1, \dots, UN_n\}, \\
\quad R(UN_i) \in \mathcal{R}_i, i = 1, \dots, n\}} \\
\\
\frac{\vdash SPR_1 \Rightarrow \mathcal{R}_1 \quad \vdash SPR_2 \Rightarrow \mathcal{R}_2 \quad \mathcal{R} = \mathcal{R}_1 ; \mathcal{R}_2}{\vdash SPR_1 \text{ then } SPR_2 \Rightarrow \mathcal{R}}
\end{array}$$

Figure 5.3: Model semantics of CASL refinements

environments that map the declared unit name to a branching assignment in the semantics of the refinement used in the declaration:

$$\frac{\Gamma_m, \Gamma_s \vdash SPR \Rightarrow \mathcal{R}}{\Gamma_s, \Gamma_m, C_s, C \vdash UN : SPR \Rightarrow C^\emptyset[UN/\mathcal{R}]}$$

Then, the units in $\pi_1(RE)$ produce a model of the result unit, which is then paired with the projection π_2 of the result-unit environment to the second component to obtain an architectural model. In the case of compositions, similarly to the static semantics, we define an auxiliary partial operation to compose refinement relations. The intuition is that at each refinement step we further restrict the domain of a constructor by compositions, thus narrowing the class of acceptable realizations, which is the image of the constructor.

Definition 5.2.6 [Mossakowski et al., 2005] *Given two refinement relations $\mathcal{R}_1, \mathcal{R}_2$ over refinement signatures $R\Sigma_1, R\Sigma_2$, respectively, such that the composition $R\Sigma = R\Sigma_1 ; R\Sigma_2$ is defined, the composition $\mathcal{R}_1 ; \mathcal{R}_2$ is defined as a refinement relation over $R\Sigma$ as follows:*

- $R\Sigma_1 = (U\Sigma, U\Sigma')$, $R\Sigma_2 = (U\Sigma', B\Sigma'')$: then $\mathcal{R}_1 ; \mathcal{R}_2$ is defined only if for all $(U', BM'') \in \mathcal{R}_2$ we have $(U, U') \in \mathcal{R}_1$ for some U . Then

$$\mathcal{R}_1 ; \mathcal{R}_2 = \{(U, BM'') \mid (U, U') \in \mathcal{R}_1, (U', BM'') \in \mathcal{R}_2 \text{ for some } U'\}$$

- $R\Sigma_1 = (U\Sigma, BstC')$ and $R\Sigma_2$ is a component refinement signature that matches $BstC'$: then $\mathcal{R}_1; \mathcal{R}_2$ is defined only if for each $R_2 \in \mathcal{R}_2$ there is $(U, BE') \in \mathcal{R}_1$ such that R_2 matches BE' , that is, for each $UN \in \text{dom}(R_2)$,
 - either $BstC'(UN)$ is a unit signature and then $R_2(UN) = (U'', BM'')$ with $U'' = BE'(UN)$, or
 - $BstC'(UN)$ is a branching static context and then $R_2(UN)$ matches $BE'(UN)$.

Then

$$\mathcal{R}_1; \mathcal{R}_2 = \{(U, BE'[R_2]) \mid (U, BE') \in \mathcal{R}_1, R_2 \in \mathcal{R}_2, R_2 \text{ matches } BE'\}$$

where given any branching environment BE' that fits $BstC'$ and assignment R_2 that matches BE' , $BE'[R_2]$ modifies BE' on each $UN \in \text{dom}(R_2)$ as follows:

- if $BstC'(UN)$ is a unit signature then $BE'[R_2](UN) = BM''$ where $R_2(UN) = (BE'(UN), BM'')$;
 - if $BstC'(UN)$ is a branching static context then we put $BE'[R_2](UN) = BE'(UN)[R_2(UN)]$.
- $R\Sigma_1$ and $R\Sigma_2$ are component refinement signatures such that for all $UN \in \text{dom}(R\Sigma_1) \cap \text{dom}(R\Sigma_2)$, $R\Sigma_{UN} = R\Sigma_1(UN); R\Sigma_2(UN)$ is defined then $\mathcal{R}_1; \mathcal{R}_2$ is defined only if for each $R_2 \in \mathcal{R}_2$ there is $R_1 \in \mathcal{R}_1$ such that R_1 transfers to R_2 , that is, for each $UN \in \text{dom}(R_1) \cap \text{dom}(R_2)$,
 - either $R\Sigma_1(UN)$ is a unit refinement signature $(U\Sigma, U\Sigma')$, and then $R_1(UN) = (U, U'_1)$ and $R_2(UN) = (U'_2, BM'')$ with $U'_1 = U'_2$, or
 - $R\Sigma_1(UN)$ is a branching refinement signature $(U\Sigma, BstC')$, where $BstC'$ is a branching unit context, and then $R_1(UN) = (U, BE')$ and $R_2(UN)$ is an assignment that matches BE' , or
 - $R\Sigma_1(UN)$ is component refinement signature, and then $R_1(UN)$ transfers to $R_2(UN)$.

Then

$$\mathcal{R}_1; \mathcal{R}_2 = \{R_1; R_2 \mid R_1 \in \mathcal{R}_1, R_2 \in \mathcal{R}_2, R_1 \text{ transfers to } R_2\}$$

where given any assignments R_1, R_2 over $R\Sigma_1, R\Sigma_2$, respectively, such that R_1 transfers to R_2 , then $R_1; R_2$ is the assignment that modifies the (ill-defined) union of R_1 and R_2 on each $UN \in \text{dom}(R_1) \cap \text{dom}(R_2)$ as follows:

- if $R\Sigma_1(UN) = (U\Sigma, U\Sigma')$, $R_1(UN) = (U, U'_1)$ and $R_2(UN) = (U'_2, BM'')$ (hence $U'_1 = U'_2$) then $(R_1; R_2)(UN) = (U, BM'')$;
- if $R\Sigma_1(UN) = (U\Sigma, BstC')$, where $BstC'$ is a branching unit context, $R_1(UN) = (U, BE')$ (hence $R_2(UN)$ is an assignment that matches BE') then $(R_1; R_2)(UN) = (U, BE'[R_2(UN)])$;

- if $R\Sigma_1(UN)$ is a component refinement signature (hence $R_1(UN)$ and $R_2(UN)$ are component assignments such that $R_1(UN)$ transfers to $R_2(UN)$) then

$$(R_1 ; R_2)(UN) = R_1(UN) ; R_2(UN).$$

Example 5.2.7 Let us apply the model semantic rules for the refinements in Ex. 5.1.3.

- With the rule for unit specifications as refinement specifications we obtain $\vdash \text{MONOID qua SPEC-REF} \Rightarrow \{(U, U) \mid U \in \text{Unit}(\text{MONOID})\}$ and similarly $\vdash \text{NAT qua SPEC-REF} \Rightarrow \{(V, V) \mid V \in \text{Unit}(\text{NAT})\}$. For R1 we must use the rule for simple refinements. For each model N of NAT, we have that $N|_\sigma \in \text{Mod}(\text{MONOID})$, where σ is the signature morphism induced by $\text{Elem} \mapsto \text{Nat}$. Thus the rule can be applied and we get that $\vdash \text{R1} \Rightarrow \{(V|_\sigma, V) \mid V \in \text{Unit}(\text{NAT})\}$;
- With a similar reasoning, we get $\vdash \text{R2} \Rightarrow \{(U|_{\sigma'}, U) \mid U \in \text{Unit}(\text{NATBIN})\}$ where σ' is the signature morphism induced by $\text{Nat} \mapsto \text{Bin}$;
- For R3 we must compose the refinement models of R1 and R2. We are in the first case of the definition of composition of refinement relations and moreover for every assignment $(B|_{\sigma'}, B)$ with $B \in \text{Unit}(\text{NATBIN})$ we have that $B|_{\sigma'}$ is a NAT-model. Then the assignment $(B|_{\sigma'}|_\sigma, B|_{\sigma'})$ is in the refinement relation of R1. We then get $\vdash \text{R3} \Rightarrow \{(B|_{\sigma'}|_\sigma, B) \mid B \in \text{Unit}(\text{NATBIN})\}$;
- Models of ADDITION_FIRST are pairs of form $(F[N], \{N \in \text{Unit}(\text{NAT}, F \in \text{Unit}(\text{NAT} \rightarrow \text{NATWITHSUC}))\})$. Note that the condition in the rule for simple refinements holds trivially in the case of R4 and R4 has the same refinement model as ADDITION_FIRST;
- With the rule for component specification refinements, we get that $\vdash \text{RCOMP} \Rightarrow \{N \mapsto \{(U|_{\sigma'}, U) \mid U \in \text{Unit}(\text{NATBIN})\}\}$;
- For R5 we must compose the refinement models of R4 and RCOMP. We are in the second case and the refinement signatures match. For any assignment $\{N \mapsto (B|_\sigma, B)\}$ with $B \in \text{Unit}(\text{NATBIN})$ we know that $B|_\sigma$ is in $\text{Unit}(\text{NAT})$ and thus there are assignments $R_1 = (U_1, BE_1)$ in the refinement relation of R4 such that $BE_1(N) = B|_\sigma$. For any such assignment R_1 we take in the composition the assignment $(U_1, BE_1[N/B])$.

Part II

HETS Development

Logical Frameworks in HETS

Contents

6.1 Preliminaries	80
6.1.1 Proof-Theoretic Logical Frameworks	80
6.1.2 A Logic Atlas in LF	82
6.2 The LATIN Metaframework	84
6.2.1 Logical Meta-Frameworks	85
6.2.2 Generalizations	89
6.3 Logical Frameworks in HETS	90
6.3.1 Implementing the LMF in HETS	90
6.3.2 LF as a Logical Framework in HETS	92
6.3.3 Adding a New Logic in HETS: FOL	92
6.4 Conclusion and Future Work	93

As already mentioned in Sec. 2.1, the logics of HETS are specified on the meta-level rather than within the system itself. Each logic or logic translation has to be specified by implementing a Haskell interface that is part of the HETS code, and tools for parsing and static analysis have to be provided. Consequently, only HETS developers but not users can add them. Besides the obvious disadvantage of the cost involved when adding logics, this representation does not provide us with a way to reason about the logics or their translations themselves. In particular, each logic’s static analysis is part of the trusted code base, and the translations cannot be automatically verified for correctness.

Moreover, HETS is based on a model-theoretical approach to logical systems – the semantics of implemented logics and the correctness of translations are determined by model theoretic arguments. Proof theory is only used as a tool to discharge proof obligations and is not represented explicitly.

In contrast, the proof theoretic approach of logical frameworks focuses on the syntactic entailment relation between sentences, which gives rise to proof terms. A sentence is then true if there exists a proof term for it. The most important proof theoretic logical frameworks are Automath [de Bruijn, 1970], Isabelle [Paulson, 1994] and the Edinburgh Logical Framework [Harper et al., 1993]. The main use of these frameworks is to encode deductive systems: object logics become thus theories in the “universal” logic of the framework. Note that using Edinburgh Logical Framework as a universal logic in which the other logics are represented has already been suggested in [Harper et al., 1994, Tarlecki, 1996].

The work reported here is part of the ongoing project LATIN (Logic Atlas and Integrator, [Kohlhase et al., 2009]). LATIN has two main goals: to *fully integrate proof and model theoretic frameworks* preserving their respective advantages, and to create *modular formalizations of commonly used logics* together with *logic morphisms interrelating them*: the **Logic Atlas**. To this end, we develop general definition of a logical framework (the **LATIN metaframework**) that covers logical frameworks such as LF, Isabelle, and rewriting logic and implement it in HETS. The LATIN metaframework follows a “logics as theories/translations as morphisms” approach such that a theory graph in a logical framework leads to a graph of institutions and comorphisms via a general construction. This means that new logics can now be added to HETS in a purely declarative way. Moreover, the declarative nature means that logics themselves are no longer only formulated in the semi-formal language of mathematics, but now are fully formal objects, such that one can reason about them (e.g. prove soundness of proof systems or logic translations) within proof systems like Twelf.

6.1 Preliminaries

6.1.1 Proof-Theoretic Logical Frameworks

We use the term *proof theoretic* to refer to logical frameworks whose semantics is or can be given in a formal and thus mechanizable way without reference to a Platonic universe. These frameworks are declarative formal languages with an inference system defining a consequence relation between judgments. They come with a notion of language extensions called signatures or theories, which admits the structure of a category. Logic encodings represent the syntax and proof theory of a logic as a theory of the logical framework, and logical consequence is represented in terms of the consequence relation of the framework.

The most important logical frameworks are LF, Isabelle, and rewriting logic. LF [Harper et al., 1993] is based on dependent type theory; logics are encoded as LF signatures, proofs as terms using the Curry-Howard correspondences, and consequence between formulas as type inhabitation. The main implementation is Twelf [Pfenning and Schürmann, 1999]. LF has been used extensively to represent logics [Harper et al., 1994, Pfenning et al., 2003, Avron et al., 1998], many of them included in the Twelf distribution. The Isabelle system [Paulson, 1994] implements higher-order logic [Church, 1940]; logics are represented as HOL theories, and consequence between formulas as HOL propositions. Logic representations in Isabelle are notable for the size of the libraries in the encoded logics, especially for HOL [Nipkow et al., 2002]. The Maude system [Clavel et al., 2007] is related to rewriting logic [Meseguer, 1992]. Here logics are represented as rewrite theories, and consequence between formulas as rewrite judgments. Logic representations in rewriting logic [Martí-Oliet and Meseguer, 1994] using the Maude system [Clavel et al., 2007] include the examples of equational logic, Horn logic and linear logic. A notable property of rewriting logic is *reflection* i.e. one can represent rewriting logic

within itself. Zermelo-Fraenkel and related set theories were encoded in a number of systems, see, e.g., [Paulson and Coen, 1993] or [Trybulec and Blair, 1985]. Other systems employed to encode logics include Coq [Bertot and Castéran, 2004], Agda [Norell, 2005], and Nuprl [Constable et al., 1986]. Only few logic *translations* have been formalized systematically in this setting. Important translations represented using the logic programming interpretation of LF include cut elimination [Pfenning, 2000] and the HOL-Nuprl translation [Schürmann and Stehr, 2004]. The latter guided the design of the Delphin system [Poswolsky and Schürmann, 2008] for logic translations.

In the following, we give an overview of **LF**, which we will use as a running example. LF extends simple type theory with dependent function types and is related to Martin-Löf type theory [Martin-Löf, 1974].

LF expressions E are grouped into kinds K , kinded type-families $A : K$, and typed terms $t : A$. The kinds are the base kind `type` and the dependent function kinds $\Pi_{x:A} K$. The type families are the constants a , applications $A t$, and the dependent function type $\Pi_{x:A} B$; type families of kind `type` are called types. The terms are constants c , variables x , applications $s t$, and abstractions $\lambda_{x:A} t$. We write $A \rightarrow B$ instead of $\Pi_{x:A} B$ if x does not occur in B .

Kinds	K	$::=$	<code>type</code> $\Pi_{x:A} K$
Type families	A, B	$::=$	a $A t$ $\Pi_{x:A} B$
Terms	s, t	$::=$	c x $s t$ $\lambda_{x:A} t$

The following grammar is a simplified version of the LF grammar where we write `·` for the empty list. It includes LF signature morphisms, which were added to LF in [Harper et al., 1994] and to Twelf in [Rabe and Schürmann, 2009]:

Signatures	Σ	$::=$	<code>·</code> $\Sigma, a : K \langle = A \rangle$ $\Sigma, c : A \langle = t \rangle$
Morphisms	σ	$::=$	<code>·</code> $\sigma, c := t$ $\sigma, a := A$

An LF **signature** Σ is a list of kinded type family declarations $a : K$ and typed constant declarations $c : A$. Both may carry definitions, i.e., $c : A = t$ and $a : K = A$, respectively; we have marked the optional definitions with $\langle \cdot \rangle$. Due to the Curry-Howard representation, propositions are encoded as types as well; hence a constant declaration $c : A$ may be regarded as an axiom A , while $c : A = t$ additionally provides a proof t for A . Hence, an LF signature corresponds to what usually is called a logical *theory*.

Relative to a signature Σ , closed expressions are related by the judgments $\vdash_{\Sigma} E : E'$ and $\vdash_{\Sigma} E = E'$. Equality of terms, type families, and kinds are defined by $\alpha\beta\eta$ -equality. All judgments for typing, kinding, and equality are decidable.

Given two signatures Σ and Σ' , an LF **signature morphism** $\sigma : \Sigma \rightarrow \Sigma'$ is a typing- and kinding-preserving map of Σ -symbols to Σ' -expressions. Thus, σ maps every constant $c : A$ of Σ to a term $\sigma(c) : \bar{\sigma}(A)$ and every type family symbol $a : K$ to a type family $\sigma(a) : \bar{\sigma}(K)$. Here, $\bar{\sigma}$ is the homomorphic extension of σ to Σ -expressions, and we will write σ instead of $\bar{\sigma}$ from now on.

Signature morphisms preserve typing, i.e., if $\vdash_{\Sigma} E : E'$, then $\vdash_{\Sigma'} \sigma(E) : \sigma(E')$, and correspondingly for kinding and equality. Due to the Curry-Howard encoding of axioms, this corresponds to theorem preservation of theory morphisms. Composition and identity are defined in the obvious way, and we obtain a category \mathbb{LF} .

In [Rabe and Schürmann, 2009], a **module system** was given for LF and implemented in Twelf. The module system permits to build both signatures and signature morphisms in a structured way. Its expressivity is similar to that of development graphs (which we described in Sec.4.3).

6.1.2 A Logic Atlas in LF

In the LATIN project [Kohlhase et al., 2009], we aim at the creation of a logic atlas based on LF. The Logic Atlas is a multi-graph of LF signatures and morphisms between them. Currently it contains formalizations of various logics, type theories, foundations of mathematics, algebra, and category theory.

Among the logics formalized in the Atlas are propositional (*PL*), first (*FOL*) and higher-order logic (*HOL*), sorted (*FOL_{ms}*) and dependent first-order logic (*DFOL*), description logics (*DL*), modal (*ML*) and common logic (*CL*) as illustrated in the diagram below. We distinguish e.g. the many-sorted first-order logic in HETS, denoted **FOL_{ms}** and its representation in the LATIN Atlas, denoted *FOL_{ms}*. Single arrows (\rightarrow) in this diagram denote LF formalizations of logic translations and hooked arrows (\hookrightarrow) denote module imports between the two logics. Among the foundations are encodings of Zermelo-Fraenkel set theory, Isabelle’s higher-order logic, and Mizar’s set theory [Iancu and Rabe, 2011].

Note that a logical framework leaves the choice of the foundation deliberately open. In this way, we can use one logical framework (e.g. LF) with several foundations (e.g. ZFC, as well as category theory). Only the representation of a logic includes the choice of a foundation.

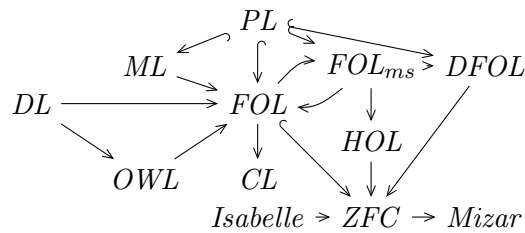
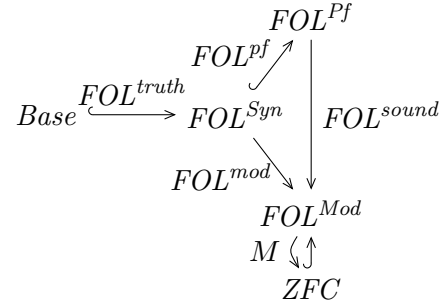


Figure 6.1: The LATIN Atlas.

Actually the graph is significantly more complex as we use the LF module system to obtain a maximally modular design of logics. For example, first-order, modal, and description logics are formed from orthogonal modules for the individual connectives, quantifiers, and axioms. For example, the \wedge connective is only declared once in the whole Atlas and imported into the various logics and foundations and

related to the type theoretic product via the Curry-Howard correspondence.

Moreover, we can use individual modules for syntax, proof theory and model theory so that the same syntax can be combined with different interpretations. For example, the LATIN formalization of first-order logic (presented in [Horozal and Rabe, 2011]) consists of the signatures $Base$ and FOL^{Syn} for syntax, FOL^{Pf} for proof theory, and FOL^{Mod} for model theory as illustrated in the diagram on the right. $Base$ contains declarations $o : \text{type}$ and $i : \text{type}$ for the type of formulas and first-order individuals, and a truth judgment for formulas. FOL^{Syn} contains declarations for all logical connectives and quantifiers (see Fig. 6.4). FOL^{truth} is an inclusion morphism from $Base$ to FOL^{Syn} . FOL^{Pf} consists of declarations for judgments and inference rules associated with each logical symbol declared in FOL^{Syn} . FOL^{pf} is simply an inclusion morphism from FOL^{Syn} to FOL^{Pf} .



For the representation of FOL model theory, LF is not a suitable metalanguage because its type theory is minimalistic and the use of higher-order abstract syntax is incompatible with the natural way of adding computational support needed to express models. However, LF can serve as a minimal, neutral framework to formalize the metalanguage itself. We choose ZFC set theory as the appropriate metalanguage because it is the standard foundation of mathematics, and formalize it in LF (in the signature ZFC) and use it as the metalanguage to define models.

The ZFC encoding includes the type of sets, the membership predicate as a primitive non-logical symbol, and the usual ZFC set operations and axioms defined in a first-order language with description operator. Additionally, ZFC contains a type judgment $elem$ for the elements of a set as well as a binary operation \implies on sets that returns the set of functions. This is important for being able to represent models as signature morphisms (see below): signature morphisms map types to types, and via $elem$, (carrier) sets can be turned into types.

FOL^{Mod} includes ZFC as a metalanguage and uses it to axiomatize the properties of FOL-models. More precisely, FOL^{Mod} declares a set $bool$ for the boolean values axiomatizing it to get the desired 2-element set $\{0, 1\}$, declares a fixed set $univ$ of individuals, along with an axiom stating that the universe is nonempty. For each logical symbol s^{Syn} in FOL^{Syn} , FOL^{Mod} declares a symbol s^{Mod} that represents the semantic operation used to interpret s^{Syn} along with axioms specifying its truth values. For instance, for disjunction, which is declared as $or : o \rightarrow o \rightarrow o$ in FOL^{Syn} , FOL^{Mod} declares the symbol \vee as a ZFC-function from $bool^2$ to $bool$ and axiomatizes it to be the binary supremum in the boolean 2-element lattice. This corresponds to the case-based definition of the semantics of a formula.

FOL^{Syn}	FOL^{Mod}	ZFC
$i : \mathbf{type}$	$univ : \mathbf{set}$	$set : \mathbf{type}$
$o : \mathbf{type}$	$bool : \mathbf{set}$	$prop : \mathbf{type}$
$or : o \rightarrow o \rightarrow o$	$\vee : elem(bool \implies bool \implies bool)$	$\vee : prop \rightarrow prop \rightarrow prop$
$forall : (i \rightarrow o) \rightarrow o$	$\forall : elem((univ \implies bool) \implies bool)$	$\forall : (set \rightarrow prop) \rightarrow prop$
		$\in : set \rightarrow set \rightarrow prop$
		$elem : set \rightarrow \mathbf{type}$
		$\implies : set \rightarrow set \rightarrow set$

The morphism FOL^{mod} interprets the syntax of FOL in the semantic realm specified by FOL^{Mod} : It maps the type i of individuals to the type of elements of $univ$, the type o of formulas to the type of elements of $bool$, and the logical operations to the corresponding operations on booleans.

The individual FOL-models are represented as LF signature morphisms from FOL^{Mod} to ZFC that are the identity on ZFC . In other words, a model M maps $univ$ to a nonempty set expressed by using the set operations of ZFC . M interprets the boolean operations in FOL^{Mod} in terms of the usual set operations in ZFC . For instance, the universal quantification for the booleans is mapped to the intersection of a family of subsets. This interpretation is ensured by the axiomatization of universal quantification. Given such a morphism M , the composition $FOL^{mod}; M$ then yields the interpretation of FOL^{Syn} in ZFC .

A particular aspect of this formalization is that soundness of FOL can be represented naturally as an LF signature morphism from FOL^{Pf} to FOL^{Mod} making the diagram above commute. Note that a morphisms in the opposite direction, i.e., from FOL^{Mod} to FOL^{Pf} , does not yield completeness.

6.2 The LATIN Metaframework

In this section we describe the theoretical background of the LATIN metaframework (LMF) based on the approach taken in [Rabe, 2010]. The LMF is an abstract framework that allows to represent logical frameworks as declarative languages given by categories of theories. The LMF is generic in the sense that it can be instantiated with specific logical frameworks such as LF, Isabelle or rewriting logic, thus allowing HETS to be flexible in the choice of the logical framework in which logics should be represented.

In Sect. 6.2.1, we show that this abstract representation of logical frameworks complies with the notion of institutions and institution comorphisms. Here we deliberately restrict attention to a special case of [Rabe, 2010] that makes the ideas clearer and discuss generalizations in Sect. 6.2.2.

6.2.1 Logical Meta-Frameworks

Definition 6.2.1 (Inclusions) A category with inclusions consists of a category together with a broad subcategory that is a partial order. We write $B \hookrightarrow C$ for the inclusion morphism from B to C .

Definition 6.2.2 (Logical Framework) A tuple $(\mathbb{C}, Base, \mathbf{Sen}, \vdash)$ is a logical framework if

- \mathbb{C} is a category that has inclusions and pushouts along inclusions,
- $Base$ is an object of \mathbb{C} ,
- $\mathbf{Sen} : \mathbb{C} \setminus Base \rightarrow Set$ is a functor, where $\mathbb{C} \setminus Base$ is the so-called slice category of \mathbb{C} over $Base$, whose objects are arrows in \mathbb{C} of source $Base$ and morphisms make triangles commute,
- for $t \in \mathbb{C} \setminus Base$, \vdash_t is a unary predicate on $\mathbf{Sen}(t)$,
- \vdash is preserved under signature morphisms: if $\vdash_t F$ then $\vdash_{t'} \mathbf{Sen}(\sigma)(F)$ for any morphism $\sigma : t \rightarrow t'$ in $\mathbb{C} \setminus Base$.

\mathbb{C} is the category of theories of the logical framework. Our focus is on declarative frameworks where theories are lists of named declarations. Typically these have inclusions and pushouts along them in a natural way.

Logics are encoded as theories Σ of the framework, but not all theories can be naturally regarded as logic encodings. Logic encodings must additionally distinguish certain objects over Σ that encode logical notions. Therefore, we consider \mathbb{C} -morphisms $t : Base \rightarrow \Sigma$ where $Base$ makes precise what objects must be distinguished.

We leave the structure of $Base$ abstract, but we require that slices $t : Base \rightarrow \Sigma$ provide at least a notion of sentences and truth for the logic encoded by Σ . Therefore, $\mathbf{Sen}(t)$ gives the set of sentences, and the predicate $\vdash_t F$ expresses the truth of F .

Example 6.2.3 (LF) We define a logical framework \mathbb{F}^{LF} based on the category \mathbb{C} to be the category of $\mathbb{L}\mathbb{F}$ signatures and signature morphisms. $\mathbb{L}\mathbb{F}$ has inclusions by taking the subset relation between sets of declarations. Given $\sigma : \Sigma \rightarrow \Sigma'$ and an inclusion $\Sigma \hookrightarrow \Sigma, c : A$, a pushout is given by

$$(\sigma, c := c) : (\Sigma, c : A) \rightarrow (\Sigma', c : \sigma(A))$$

(except for possibly renaming c if it is not fresh for Σ'). The pushouts for other inclusions are obtained accordingly.

$Base$ is the signature with the declarations $o : \mathbf{type}$ and $\mathbf{ded} : o \rightarrow \mathbf{type}$. For every slice $t : Base \rightarrow \Sigma$, we define $\mathbf{Sen}(t)$ as the set of closed $\beta\eta$ -normal LF-terms of type $t(o)$ over the signature Σ . Moreover, $\vdash_t F$ holds iff the Σ -type $t(\mathbf{ded}) F$ is inhabited.

Given $t : Base \rightarrow \Sigma$ and $t' : Base \rightarrow \Sigma'$ and $\sigma : \Sigma \rightarrow \Sigma'$ such that $t\sigma = t'$, we define the sentence translation by $\mathbf{Sen}(\sigma)(F) = \sigma(F)$. Truth is preserved: assume $\vdash_t F$; thus $t(\text{ded}) F$ is inhabited over Σ ; then $\sigma(t(\text{ded}) F) = t'(\text{ded}) \sigma(F)$ is inhabited over Σ' ; thus $\vdash_{t'} \mathbf{Sen}(\sigma)(F)$.

We present the integration of \mathbb{LF} in HETS in Sec. 6.3.2.

Example 6.2.4 (Isabelle) A logical framework based on Isabelle is defined similarly. \mathbb{C} is the category of Isabelle theories and theory morphisms (for the latter, see [Bortin et al., 2006]). Base consists of the declarations $bool : type$ and $\text{trueprop} : bool \rightarrow \text{prop}$ where prop is the type of Isabelle propositions. Given $t : Base \rightarrow \Sigma$, we define $\mathbf{Sen}(t)$ as the set of Σ -terms of type $t(bool)$, and $\vdash_t F$ holds if $t(\text{trueprop}) F$ is an Isabelle theorem over Σ .

Note that Isabelle is already available in HETS.

Example 6.2.5 (Rewriting logic) A logical framework based on rewriting logic can be defined along the lines of [Martí-Oliet and Meseguer, 1994], using the system Maude. \mathbb{C} is the category of rewriting logic theories and theory morphisms. Base consists of the following Maude declarations:

```

sorts Prop FormList Sequent .
subsorts Prop < FormList .
op empty : -> FormList .
op tt : -> Prop .
op __ ⊢ __ : FormList FormList -> Sequent .

```

where Prop stands for the type of propositions, tt for the formula *true*, and \vdash turns two lists of formulas into a sequent. Given $t : Base \rightarrow \Sigma$, we define $\mathbf{Sen}(t)$ as the set of Σ -terms of type $t(\text{Prop})$, and $\vdash_t F$ holds for some term F of type $t(\text{Prop})$ if $\text{empty} \vdash F \Rightarrow_{\Sigma} \text{empty} \vdash \text{tt}$. \vdash_t is preserved by rewriting logic theory morphisms because rewriting must be preserved.

We present the Maude logic and the integration of Maude in HETS in Chapter 9.

Logical frameworks are then used to define institutions. The basic idea is that slices $t : Base \rightarrow L^{Syn}$ define logics (L^{Syn} specifies the syntax of the logic), signatures of that logic are extensions $L^{Syn} \hookrightarrow \Sigma^{Syn}$, and sentences and truth are given by \mathbf{Sen} and \vdash . We could represent the logic's models in terms of the models of the logical framework, but that would complicate the mechanizable representation of models. Therefore, we represent models as \mathbb{C} morphisms into a fixed theory that represents the foundation of mathematics. We need one auxiliary definition to state this precisely:

Definition 6.2.6 Fix a logical framework, and assume $L^{mod} : L^{Syn} \rightarrow L^{Mod}$ in \mathbb{C} as in the diagram below.

$$\begin{array}{ccccc}
& & \curvearrowright & & \\
L^{Mod} & \hookrightarrow & \Sigma^{Mod} & \xrightarrow{\sigma^{mod}} & \Sigma'^{Mod} \\
\uparrow L^{mod} & & \uparrow \Sigma^{mod} & & \uparrow \Sigma'^{mod} \\
L^{Syn} & \hookrightarrow & \Sigma^{Syn} & \xrightarrow{\sigma^{syn}} & \Sigma'^{Syn} \\
& & \curvearrowleft & &
\end{array}$$

Firstly, for every inclusion $L^{Syn} \hookrightarrow \Sigma^{Syn}$, we define Σ^{Mod} and Σ^{mod} such that Σ^{Mod} is a pushout. Secondly, for every $\sigma^{syn} : \Sigma^{Syn} \rightarrow \Sigma'^{Syn}$, we define $\sigma^{mod} : \Sigma^{Mod} \rightarrow \Sigma'^{Mod}$ as the unique morphism such that the above diagram commutes.

We can now give the main definition:

Definition 6.2.7 (Institutions in LMF) Let $\mathbb{F} = (\mathbb{C}, Base, \mathbf{Sen}, \vdash)$ be a logical framework. Assume $L = (L^{Syn}, L^{truth}, L^{Mod}, \mathcal{F}, L^{mod})$ as in the following diagram:

$$\begin{array}{ccccccc}
& & \mathcal{F} & \xrightarrow{id_{\mathcal{F}}} & \mathcal{F} & \xleftarrow{m'} & \\
& & \downarrow & & \uparrow m & & \\
& & L^{Mod} & \hookrightarrow & \Sigma^{Mod} & \xrightarrow{\sigma^{mod}} & \Sigma'^{Mod} \\
& & \uparrow L^{mod} & & \uparrow \Sigma^{mod} & & \uparrow \Sigma'^{mod} \\
Base & \xrightarrow{L^{truth}} & L^{Syn} & \hookrightarrow & \Sigma^{Syn} & \xrightarrow{\sigma^{syn}} & \Sigma'^{Syn}
\end{array}$$

Then we define the institution $\mathbb{F}(L) = (\mathbf{Sig}^L, \mathbf{Sen}^L, \mathbf{Mod}^L, \models^L)$ as follows:

- \mathbf{Sig}^L is the full subcategory of $\mathbb{C} \setminus L^{Syn}$ whose objects are inclusions. To simplify the notation, we will write Σ^{Syn} for an inclusion $L^{Syn} \hookrightarrow \Sigma^{Syn}$ below.
- \mathbf{Sen}^L is defined by

$$\mathbf{Sen}^L(\Sigma^{Syn}) = \mathbf{Sen}(L^{truth}; (L^{Syn} \hookrightarrow \Sigma^{Syn})) \quad \text{and} \quad \mathbf{Sen}^L(\sigma) = \mathbf{Sen}(\sigma).$$

- \mathbf{Mod}^L is defined by

$$\begin{aligned}
\mathbf{Mod}^L(\Sigma^{Syn}) &= \{m : \Sigma^{Mod} \rightarrow \mathcal{F} \mid (\mathcal{F} \hookrightarrow \Sigma^{Mod}); m = id_{\mathcal{F}}\} \\
\mathbf{Mod}^L(\sigma^{syn})(m') &= \sigma^{mod}; m'.
\end{aligned}$$

All model categories are discrete.

- We make the following abbreviation: For a model $m \in \mathbf{Mod}^L(\Sigma^{Syn})$, we write \bar{m} for $L^{truth}; (L^{Syn} \hookrightarrow \Sigma^{Syn}); \Sigma^{mod}; m : Base \rightarrow \mathcal{F}$. Then we define satisfaction by

$$m \models_{\Sigma^{Syn}}^L F \quad \text{iff} \quad \vdash_{\bar{m}} \mathbf{Sen}(\Sigma^{mod}; m)(F).$$

Theorem 6.2.8 (Institutions in LMF) *In the situation of Def. 6.2.7, $\mathbb{F}(L)$ is an institution.*

Proof. We need to show the satisfaction condition. So assume $\sigma^{syn} : \Sigma^{Syn} \rightarrow \Sigma'^{Syn}$, $F \in \mathbf{Sen}^L(\Sigma^{Syn})$, and $m' \in \mathbf{Mod}^L(\Sigma'^{Syn})$. First observe that $\overline{m'} = L^{truth}; (L^{Syn} \hookrightarrow \Sigma'^{Syn}); \Sigma'^{mod}; m' = L^{truth}; (L^{Syn} \hookrightarrow \Sigma^{Syn}); \Sigma^{mod}; (\sigma^{mod}; m') = \sigma^{mod}; m'$. Then $\mathbf{Mod}^L(\sigma)(m') \models_{\Sigma^{Syn}}^L F$ iff $\vdash_{\sigma^{mod}; m'} \mathbf{Sen}(\Sigma^{mod}; (\sigma^{mod}; m'))(F)$ iff $\vdash_{\overline{m'}} \mathbf{Sen}(\Sigma'^{mod}; m')(\mathbf{Sen}(\sigma^{syn})(F))$ iff $m' \models_{\Sigma'^{Syn}}^L \mathbf{Sen}^L(\sigma^{syn})(F)$.

Example 6.2.9 (FOL) *We can now obtain an institution from the encoding of first-order logic in Sect. 6.1.2 based on the logical framework \mathbb{F}^{LF} . First-order logic is encoded as the tuple $FOL = (FOL^{Syn}, FOL^{truth}, FOL^{Mod}, ZFC, FOL^{mod})$ as in Sect. 6.1.2.*

We obtain an institution comorphism $\mathbf{FOL} \rightarrow \mathbb{F}^{LF}(FOL)$ as follows. Signatures of \mathbf{FOL} are mapped to the extension of FOL^{Syn} with declarations $f : i \rightarrow \dots \rightarrow i \rightarrow i$ for function symbols f , $p : i \rightarrow \dots \rightarrow i \rightarrow o$ for predicate symbols p . If we want to map \mathbf{FOL} theories as well, we add declarations $ax : \text{ded } F$ for every axiom F . Signature morphisms are mapped in the obvious way. The sentence translation is an obvious bijection. The model translation maps every $m : \Sigma^{Mod} \rightarrow \mathcal{F}$ to the model whose universe is given by $m(\text{univ})$ and which interprets symbols f and p according to $m(f)$ and $m(p)$. The model translation is not surjective as there are only countably many morphisms m in $\mathbb{F}^{LF}(FOL)$. However, since \mathbf{FOL} has a constructive existence proof of canonical models, these models can be represented as ZFC terms and are in the image of the model translation. The satisfaction condition can be proved by an easy induction. $\mathbb{F}^{LF}(FOL)$ is complete thus \mathbf{FOL} and $\mathbb{F}^{LF}(FOL)$ have the same consequence relation.

Logical frameworks can also be used to encode institution comorphisms in an intuitive way:

Theorem 6.2.10 (Institution Comorphisms in LMF) *Fix a logical framework $\mathbb{F} = (\mathbb{C}, \text{Base}, \mathbf{Sen}, \vdash)$. Assume two logics $L = (L^{Syn}, L^{truth}, L^{Mod}, \mathcal{F}, L^{mod})$ and $L^{pf} = (L'^{Syn}, L'^{truth}, L'^{Mod}, \mathcal{F}, L'^{mod})$. Then a comorphism $\mathbb{F}(L) \rightarrow \mathbb{F}(L^{pf})$ is induced by morphisms (l^{syn}, l^{mod}) if the following diagram commutes*

$$\begin{array}{ccc}
 & & \mathcal{F} \\
 & \swarrow & \searrow \\
 L^{Mod} & \xrightarrow{l^{mod}} & L'^{Mod} \\
 \uparrow L^{mod} & & \uparrow L'^{mod} \\
 L^{Syn} & \xrightarrow{l^{syn}} & L'^{Syn} \\
 & \swarrow L^{truth} \quad \searrow L'^{truth} & \\
 & \text{Base} &
 \end{array}$$

Proof. A signature $L^{Syn} \hookrightarrow \Sigma^{Syn}$ is translated to $L'^{Syn} \hookrightarrow \Sigma'^{Syn}$ by pushout along l^{syn} yielding $\sigma^{syn} : \Sigma^{Syn} \rightarrow \Sigma'^{Syn}$. Sentences are translated by applying σ^{syn} . So

obtain $\sigma^{mod} : \Sigma^{Mod} \rightarrow \Sigma^{Mod}$ as the unique morphism through the pushout Σ^{Mod} . Then models are translated by composition with σ^{mod} . We omit the details.

It is easy to see that comorphisms that are embeddings can be elegantly represented in this way, as well as many inductively defined encodings. However, the assumptions of this theorem are too strong to permit the encoding of some less trivial comorphisms. For example, non-compositional sentence translations, which come up when translating modal logic to first-order logic, cannot be represented as signature morphisms. Or signature translations that do not preserve the number of non-logical symbols, which come up when translating partial to total function symbols, often cannot be represented as pushouts. More general constructions for the special case of LF are given in [Rabe, 2010] and [Sojakova, 2010].

6.2.2 Generalizations

In Ex. 6.2.9, we do not obtain a comorphism in the opposite direction. There are three reasons for that. Firstly, $\mathbb{F}^{LF}(FOL)$ contains a lot more signatures than needed because the definition of \mathbf{Sig}^L permits any extension of L^{Syn} , not just the ones corresponding to function and predicate symbols. Secondly, the discrete model categories of $\mathbb{F}^{LF}(FOL)$ cannot represent the model morphisms of **FOL**. Thirdly, only a (countable) subclass of the models of **FOL** can be represented as \mathbb{LF} morphisms. Moreover, Def. 6.2.2 and 6.2.7 are restricted to institutions, i.e., the syntax and model theory of a logic, and exclude the proof theory. We look at these problems below.

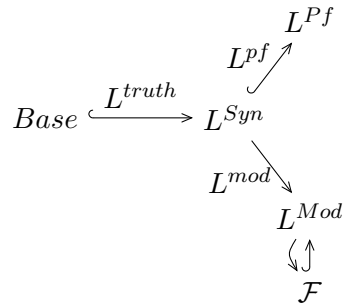
Signatures In order to solve the first problem we need to restrict $\mathbb{F}(L)$ to a subcategory of \mathbf{Sig}^L . However, it is difficult to single out the needed subcategory in a mechanizable way. A solution is provided by the MMT expression patterns [Horozal, 2012], which give a way to pattern-match declarations of the logical framework. If a concrete logic definition contains a set P of patterns, we represent its logical signatures as \mathbb{C} -objects Σ^{Syn} that extend L^{Syn} only with declarations matching one of the patterns in P .

Model Morphisms Regarding the second problem, if \mathbb{C} is a 2-category, we can define the model morphisms of $\mathbb{F}(L)$ as 2-cells in \mathbb{C} . However it is difficult in practice to obtain 2-categories for type theories such as LF or Isabelle. [Sojakova, 2010] gives a syntactical account of logical relations that behave like 2-cells in sufficiently many ways to yield model morphisms.

Undefinable Models The third problem is the most fundamental one because no formal logical framework can ever encode all models of a Platonic universe. Our encoding of ZFC is strong enough to encode any **definable** model. We call a model M definable if it arises as the solution to a formula $\exists^1 M.F(M)$ for some parameter-free formula $F(x)$ of the first-order language of ZFC. This restriction is philosophically

serious but in our experience not harmful in practice. Indeed, if infinite LF signatures are allowed, using canonical models constructed in completeness proofs, in many cases *all* models can be represented up to elementary equivalence.

Proof Theory The examples from Sect. 6.1.2 already encoded the proof theory of first-order logic in a way that treats proof theory and model theory in a balanced way. The definitions can be easily generalized to this setting, see for example [Rabe, 2010]. The outcome is in this case a diagram of the form



Our HETS representation covers this entire diagram.

6.3 Logical Frameworks in HETS

The differences between LF and HETS mentioned in Sect. 6.1 exhibit complementary strengths, and a major goal of our work is to combine them. We have enhanced HETS with a component that allows declarative definitions of new logics. The user specifies a logic by giving the representation of its constituents in a logical framework and the combined system recognizes the new logic and integrates it into the HETS logic graph. The implementation follows the HETS principles of high abstraction and separation of concerns: we provide an implementation for the general concept of logical frameworks, which we describe in Sect. 6.3.1. This is further instantiated for the particular case of LF in Sect. 6.3.2. Finally, in Sect. 6.3.3 we present a complete description of the steps necessary to add a new logic in HETS using the framework of LF.

6.3.1 Implementing the LMF in HETS

We now present how the concept of logical framework is integrated into HETS. Note that this is done on the developer's side and it is thus not visible to the user. Once the integration of a new logical framework is done, the user can use it as a meta-logic for the object logics he wants to specify.

Recall that in Sec. 2.1 we presented the Haskell type class `Logic`, which provides an interface for implementing new logics in HETS. Similarly, the central part of the implementation is a Haskell type class `LogicalFramework`, which is then instantiated by the logics which can be used as logical frameworks, i.e. in which

object logics can be specified by the user. As we have seen, such candidates are LF, rewriting logic and Isabelle; currently only LF has a full implementation in HETS as a logical framework. The class `LogicalFramework` provides a selector `base_sig` for the *Base* signature of the logical framework, a method `write_logic` which takes an object logic name as an argument and generates Haskell source code with the instance of class `Logic` for the given object logic and stores the morphisms L^{truth} , L^{pf} and L^{mod} respectively in their internal Haskell representation. Note that L^{truth} is needed in the static analysis of the specifications written in the object logic, while L^{pf} and L^{mod} are needed to check correctness of logic translations.

Since we can only instantiate this type class for logics, we get the category \mathbb{C} mentioned in Def. 6.2.2 from the requirement that signatures and signature morphisms of a logic provide an instance of the type class `Category`. The sentence functor **Sen** is specified implicitly by the `write_logic` method: the instantiation of the `StaticAnalysis` class determines exactly which sentences are valid for a particular signature of L , thus giving **Sen** on objects. Since the current implementation of logics in HETS does not include satisfaction of sentences in models, the predicate \vdash_t is currently not represented as its main purpose is to define the satisfaction relation for object logics.

At the syntactic level, we must provide a way to write down new logic definitions in HetCASL, the underlying heterogenous algebraic specification language of HETS. Since definitions of new logics have a different status than usual algebraic specifications, we extend the language at the library level.

Concrete Syntax We add the following concrete syntax (on the right) to HetCASL in order to define new logics. Here L is the name of the newly defined logic and \mathbb{F} is an identifier pointing to the logical framework used. The identifiers L^{truth} , L^{mod} , L^{pf} , \mathcal{F} are the components of the new logic L . They refer to previously declared signature morphisms of \mathbb{F} and the signatures representing L^{Syn} , L^{Mod} , L^{Pf} can be inferred from them. \mathcal{F} is a signature which gives the foundation.

```
newlogic L =
  meta  $\mathbb{F}$ 
  syntax  $L^{truth}$ 
  models  $L^{mod}$ 
  foundation  $\mathcal{F}$ 
  proofs  $L^{pf}$ 
```

After encountering a `newlogic` declaration, HETS invokes a static analyzer, which retrieves the signatures and morphisms constituting the components of the logic L . The analyzer verifies the correct shape of the induced diagram and instantiates the `Logic` class for the logic L as specified by the `write_logic` method of the framework \mathbb{F} .

The logic L arising from the above `newlogic L` declaration differs slightly from the one described in Def. 6.2.7 in that it uses signatures of \mathbb{F} that extend L^{Syn} rather than \mathbb{F} -inclusion morphisms out of L^{Syn} . Accordingly, the morphisms of L are those morphisms of \mathbb{F} which are the identity on L^{Syn} . This is essentially the same thing, but has the advantage that the data types representing the signatures and morphisms of \mathbb{F} can be directly reused for L and no separate instantiation of

the class *Category* is required ¹.

6.3.2 LF as a Logical Framework in HETS

In this section we outline how to turn LF into a logical framework in HETS, i.e. how to instantiate the `LogicalFramework` class for LF. In order to do so we will make use of the instance of the `Logic` class for $\mathbb{L}\mathbb{F}$. An institution for LF can be defined as for example in [Rabe, 2008]. The first step is to integrate it in HETS.

A special aspect of this integration is that Twelf will be used as an intermediate tool. After receiving the input file, Twelf performs parsing, static analysis and reconstruction of types and implicit arguments. If the analysis succeeds, the output is stored as an OMDoc version of the input file, and is subsequently imported into HETS using standard XML technologies. HETS reads the imported OMDoc file and transforms it into corresponding LF signatures and morphisms in their HETS internal representation.

We now come to instantiating the class `LogicalFramework` for LF. The *Base* signature is specified to be the LF signature containing the symbols *o* and *ded*, as described in Ex. 6.2.3. The instantiations of the classes `Logic`, `Syntax`, etc. provided by the `write_logic` method mostly inherit their LF implementations, with one exception being the `StaticAnalysis` class. While both LF and the LF object logics use Twelf to verify the well-formedness of input specifications, a specification in an object logic *L* is assumed to have been given relative to the L^{Syn} signature supplied when defining the object logic *L* and stored as target of the morphism L^{truth} .

6.3.3 Adding a New Logic in HETS: FOL

We will now illustrate the steps needed to add first-order logic as a new logic in HETS. The aim of this section is not to show how to encode a particular logic in Twelf, which for the case of first-order logic has been described in [Horozal and Rabe, 2011], but rather to show how an existing encoding can be used to add the logic in HETS.

Given a *FOL* encoding as in Section 6.1.2, all that is needed to be done is to collect the components of the encoding in a `newlogic` definition, as in Fig. 6.2. The first lines import the morphism FOL^{truth} from *Base* to FOL^{Syn} , the morphism FOL^{mod} from FOL^{Syn} to FOL^{Mod} , and the morphism FOL^{Pf} from FOL^{Syn} to FOL^{Pf} as in Ex. 6.2.9, from their respective directories. *STTIFOLEQ* is a fragment of *ZFC* used to represent model theory. It is composed of simple type theory equipped with external intuitionistic first-order logic. We assume for convenience that the file with the new logic definition is in the folder that contains the directory of logics as sub-

¹The theory presented in Section 6.2 could thus have been formulated equivalently, albeit less elegantly, without referring to slice categories.

```

%%FOLtruth
from logics/first-order/syntax/fol get FOL_truth
%%FOLmod
from logics/first-order/model_theory/fol get FOL_mod
%%FOLpf
from logics/first-order/proof_theory/fol get FOL_pf
%% $\mathcal{F}$ 
from logics/meta/sttifol get STTIFOLEQ

newlogic FOL =
  meta  $\mathbb{LF}$ 
  syntax FOL_truth
  models FOL_mod
  foundation STTIFOLEQ
  proofs FOL_pf
end

```

Figure 6.2: Defining *FOL* as a new object logic.

folder; the paths need to be adjusted if that is not the case². The directory structure mirrors the modular design of logics in the Logic Atlas. As a result of calling HETS on the above file, a new directory called *FOL* is added to the source folder of HETS. The directory contains automatically generated files with the class instances needed for the logic *FOL*. Moreover, the HETS variable containing the list of available logics is updated to include *FOL*. After recompiling HETS, the new logic is added to the logic graph of HETS and can be used in the same way as any of the built-in logics.

In particular, we can now use the new object logic to write specifications. For example, the specification in Fig. 6.3 uses *FOL* as a current logic and declares a constant symbol *c* and a predicate *p*, together with an axiom that the predicate *p* holds for the constant *c*. The syntax for logics specified in a logical framework \mathbb{F} is inherited from the framework (in our case \mathbb{LF}), but it has been extended with support for sentences, in the usual CASL syntax i.e. prefixed by the '.' character.

Fig. 6.4 presents the theory of SP as displayed from within HETS; as mentioned in Section 6.3.2, the theory is automatically assumed to extend FOL^{Syn} . Since in HETS all imports are internally flattened, the theory of SP when displayed will include all the symbols from FOL^{Syn} .

6.4 Conclusion and Future Work

We have described a prototypical integration of HETS with logical frameworks in general and LF and the Twelf tool in particular. The structuring language used

²The complete specification of *FOL* in LF can be found at <https://svn.omdoc.org/repos/latin/twelf-r1687/>.

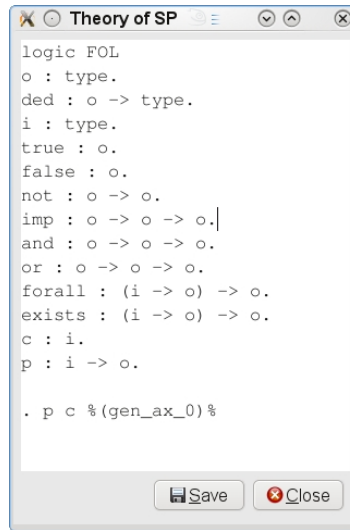
```

logic FOL
spec SP =
  c : i.
  p : i -> o.

  . p c
end

```

Figure 6.3: Specification in the new object logic.



```

logic FOL
o : type.
ded : o -> type.
i : type.
true : o.
false : o.
not : o -> o.
imp : o -> o -> o.
and : o -> o -> o.
or : o -> o -> o.
forall : (i -> o) -> o.
exists : (i -> o) -> o.
c : i.
p : i -> o.

. p c %(gen_ax_0)%

```

Figure 6.4: Theory of SP

by HETS has a model theoretic semantics, which has been reflected in the proof theoretic logical framework LF by representing models as theory morphisms into some foundation. While LF is the logical framework of our current choice, both the theory and the implementation are so general that other frameworks like Isabelle can be used as well. We expect important synergy effects from this as Isabelle is already used as one of the main inference engines in HETS.

Proof theory of the represented logics has been treated only superficially in the present work, but in fact, we have represented proof calculi for all the LATIN logics within LF. Representing models in the system as well has enabled us to formally prove soundness of the calculi. It is straightforward to extend the construction of institutions out of logic representations in logical frameworks such that they deliver institutions with proofs. In the long run, we envision that the provers integrated in HETS also return proof terms, which HETS can then fill into the original file and rerun Twelf on it to validate the proof. Thus, HETS becomes the mediator that orchestrates the interaction between external theorem provers and Twelf as a trusted proof checker.

Acknowledgement. The results of this chapter have been published as [Code-scu et al., 2010a]. The LATIN atlas (Sec. 6.1.2) was developed at Jacobs University Bremen by F. Rabe, F. Horozal and their students. The theoretical background (Sec. 6.2) is due to F. Rabe and has the origin in [Rabe, 2008]. The integration of logical frameworks in HETS (Sec. 6.3) has been done by myself jointly with K. Sojakova.

Generalized Theoroidal Comorphisms

Contents

7.1 Theoroidal Comorphisms	95
7.2 Motivating Examples	97
7.3 Generalized Comorphisms	101
7.4 Heterogeneous Specifications	102
7.5 Heterogeneous Proofs	104
7.6 Conclusions	106

Several notions of translations between institutions have been proposed, capturing different concepts; among them, institution comorphisms, also called representations or maps of institutions, which usually represent logic encodings or inclusions. This formalization is used in HETS, as the conditions for lifting the properties from the logic-specific level to the logic-independent one are in this case easier to meet in practice; however, other kind of logic translation can also be added [Mossakowski, 2003]. Comorphisms, in their theoroidal variant, are required to map theories of the same signature (i.e. using the same symbols) to theories over the same signature and this mapping must interact well with the translation of sentences. We present several logic translations of practical importance encountered in HETS that do not have this properties. This motivates us to investigate, building on the ideas in [Mossakowski, 1996], in which conditions the notion of comorphism can be generalized by dropping these restrictions and what the impact of this generalization at the heterogeneous level is.

7.1 Theoroidal Comorphisms

Let us recall from Section 3.2 that given two institutions I and J , an institution comorphism $\rho : I \rightarrow J$ consists of a functor Φ mapping I -signatures to J -signatures, a natural transformation α such that for each I -signature Σ , α_Σ maps Σ -sentences (in I) to $\Phi(\Sigma)$ -sentences (in J) and a natural transformation β such that for each I -signature Σ , β_Σ reduces $\Phi(\Sigma)$ -models (in J) to Σ -models (in I) and the following satisfaction condition must hold:

$$M' \models_{\Phi\Sigma}^J \alpha_\Sigma(e) \iff \beta_\Sigma(M') \models_\Sigma^I e$$

for each I -signature Σ , each $\Phi(\Sigma)$ -model M' (in J) and each Σ -sentence e (in I).

As noticed in [Meseguer, 1989], it is often a natural case that the signatures of the source logic are translated to theories of the target one rather than just signatures. This leads to a generalization of the concept of institution comorphism, called theoroidal comorphism in [Goguen and Roşu, 2002] and map of institutions in [Meseguer, 1989]. Firstly, the following notion captures the idea that a functor between categories of theories of two institutions behaves nicely with logical consequence. Note that in [Meseguer, 1989] the concept is introduced in the more general setting of entailment systems.

Definition 7.1.1 *Given two institutions $I = (\mathbf{Sig}^I, \mathbf{Sen}^I, \mathbf{Mod}^I, \models^I)$ and $J = (\mathbf{Sig}^J, \mathbf{Sen}^J, \mathbf{Mod}^J, \models^J)$, a functor $\Phi : \mathbb{T}h^I \rightarrow \mathbb{T}h^J$ and a natural transformation $\alpha : \mathbf{Sen}^I \rightarrow \Phi; \mathbf{Sen}^J$, we say that Φ is α -sensible if*

- *there is a functor $\Phi_0 : \mathbf{Sig}^I \rightarrow \mathbf{Sig}^J$ such that the square*

$$\begin{array}{ccc} \mathbb{T}h^I & \xrightarrow{\Phi} & \mathbb{T}h^J \\ \text{sign}^I \downarrow & & \downarrow \text{sign}^J \\ \mathbf{Sig}^I & \xrightarrow{\Phi_0} & \mathbf{Sig}^J \end{array}$$

commutes, where $\text{sign} : \mathbb{T}h \rightarrow \mathbf{Sig}$ denotes the forgetful functor and

- *for each I -theory $T = (\Sigma, \Gamma)$, if we denote $\Phi(T) = (\Sigma', \Gamma')$ and $\Phi((\Sigma, \emptyset)) = (\Sigma', \emptyset')$,*

$$(\Gamma')^\bullet = (\emptyset' \cup \alpha_\Sigma(\Gamma))^\bullet$$

This intuitively means that an α -sensible functor Φ maps theories over to same signature to theories over the same signature (the first condition) in a modular way, which means that the translation an I -theory (Σ, Γ) to J is equivalently obtained by first translating (Σ, \emptyset) to J and then adding $\alpha_\Sigma(\Gamma)$ to the resulting theory as axioms (the second condition).

A *theoroidal comorphism* between two institutions is then defined as a regular comorphism between their corresponding institutions of theories such that the theory translation is α -sensible.

Definition 7.1.2 *Given two institutions $I = (\mathbf{Sig}^I, \mathbf{Sen}^I, \mathbf{Mod}^I, \models^I)$ and $J = (\mathbf{Sig}^J, \mathbf{Sen}^J, \mathbf{Mod}^J, \models^J)$, a theoroidal comorphism $\rho = (\Phi, \alpha, \beta) : I \rightarrow J$ consists of a functor $\Phi : \mathbb{T}h^I \rightarrow \mathbb{T}h^J$, a natural transformation $\alpha : \mathbf{Sen}^I \rightarrow \Phi; \mathbf{Sen}^J$ such that Φ is α -sensible and a natural transformation $\beta : \Phi^{op}; \mathbf{Mod}^J \rightarrow \mathbf{Mod}^I$ such that the following satisfaction condition holds:*

$$M' \models_{\text{sign}(\Phi(\Sigma, \emptyset))}^J \alpha_\Sigma(e) \iff \beta_{(\Sigma, \emptyset)}(M') \models_\Sigma^I e$$

for each I -signature Σ , each $\Phi((\Sigma, \emptyset))$ -model M' (in J) and each Σ -sentence e (in I).

7.2 Motivating Examples

We will now present several logic translations in the graph of logics of Hets which do not respect the α -sensitivity requirement. To ease understanding, we will use examples instead of defining the comorphisms and the involved institutions in full detail.

Example 7.2.1 *The comorphism $CASL2SubCFOL$ encoding partiality with the help of bottom or 'undefined' elements, described as translation (5a') in [Mossakowski, 2002b] acts at the level of signatures by introducing for each sort s such that there is a term of sort s using a partial function or a projection:*

- a bottom element $bot_s : s$
- a definedness predicate $defined_s : s$,
- total function symbols for projections to subsorts $proj_{s,s'} : s \rightarrow s'$ for each subsort s' of s (when such subsorts exist)

and by turning the partial functions into total ones. Moreover, axioms are introduced for expressing the undefinedness of the bottom element, the non-emptiness of each sort, injectivity of projection and that projection maps elements identically from the supersort to the subsort.

However, the resulting theory may still contain many symbols that are not actually needed for the proofs, so we try to optimize this translation by mapping an entire theory to $SubCFOL$ and introduce symbols for encoding partiality depending on its sentences. Namely, the set of sorts for which there exists a partial term is computed considering only the subsort projections on those subsorts for which there is a sentence in the theory with a membership or a cast on the subsort, and then adding all their supersorts. The motivation for making this simplification is that provers are more efficient on smaller theories. Notice however that the theory translation component of the comorphism is obviously not signature preserving, so the α -sensitivity requirement is not met.

spec SP =

sorts $Nat < Int; Car < Vehicle$

ops $speed_limit : Vehicle \rightarrow Int;$

$car_speed_limit : Int$

• $\forall v : Vehicle$

• $v \in Car \Rightarrow speed_limit(v) = car_speed_limit$

end

Figure 7.1: First-order specification of vehicles

Let us consider the specification from Fig. 7.1, where the axiom tests whether a *Vehicle* is a *Car*. Then the projection from *Vehicle* to *Car* is considered for determining the partial terms. Note that $speed_limit(v)$ can be undefined if v is the bottom

```

spec SPEC =
  sorts Nat < Int; Car < Vehicle
  op   car_speed_limit : Int
  op   gn_bottom_Car : Car
  op   gn_bottom_Int : Int
  op   gn_bottom_Vehicle : Vehicle
  op   gn_proj_Vehicle_Car : Vehicle → Car
  op   speed_limit : Vehicle → Int
  pred gn_defined : Car; pred gn_defined : Int; pred gn_defined : Vehicle
  ∀ x, y : Vehicle
  • gn_defined(gn_proj_Vehicle_Car(x))
    ∧ gn_defined(gn_proj_Vehicle_Car(y))
    ∧ gn_proj_Vehicle_Car(x) = gn_proj_Vehicle_Car(y)
    ⇒ x = y
                                                                    %injectivity

  ∀ x : Car
  • gn_defined(x) ⇒ gn_proj_Vehicle_Car(x) = x
                                                                    %projection

  • ∃ x : Car • gn_defined(x)
                                                                    %non-empty sort

  ∀ x : Car
  • ¬ gn_defined(x) ⇔ x = gn_bottom_Car
                                                                    %undefinedness of bottom

  ∀ v : Vehicle
  • gn_defined(v)
    ⇒ gn_defined(gn_proj_Vehicle_Car(v))
    ⇒ speed_limit(v) = car_speed_limit
                                                                    %translated sentence

```

Figure 7.2: Translated signature and axioms illustrating the effect of the translation on sort Car.

element on sort *Vehicle*, so *Int* gets a bottom as well. However, no membership or cast involves the sort *Nat*, so it shall not get a bottom element. Fig. 7.2 presents the resulting signature and also the axioms introduced by the translation for the sort *Car*, with an explanatory comment for each of them.

One can also decompose this translation as the composition of the original comorphism with an endo-translation on *SubCFOL*, making the simplification. As this endo-translation maps theories in a formula-dependent way, it fails to be a comorphism.

Example 7.2.2 In [Lüttich and Mossakowski, 2007] a comorphism from CASL to

SoftFOL (an untyped variant of first-order logic with sort generation constraints, details omitted) is introduced with the purpose of connecting CASL to theorem provers. The resulting *SoftFOL* theory is translated to provers' input format; however, existing provers like SPASS do not provide support for inductive datatypes. Recovering induction proofs can be done, as explained also in [Lüttich and Mossakowski, 2007], by instantiating the induction principles corresponding to sort generation constraints for each given proof goal. Note that lemmas still have to be provided by the user.

```

spec NAT =
  free type Nat ::= 0 | suc(Nat)
end

spec COMMPLUS =
  NAT
then op   __+__ : Nat × Nat → Nat
  vars   x, y : Nat
  • 0 + y = y
  • suc(x) + y = suc(x + y)
  • 0 + x = x + 0
  • suc(x) + y = x + suc(y)
  • x + y = y + x
end

```

%implied
%implied
%implied

Figure 7.3: Specification of natural numbers, with goals marked as implied

For example, consider the specification of natural numbers as a free type generated by 0 and successor and assume we want to prove commutativity of +, using two lemmas, as in Fig. 7.3. Then Fig. 7.4 presents the axioms introduced by translation CASL2SoftFOLInduction, which extends CASL2SoftFOL as described above.

This translation is preserving signatures when mapping theories, but, since new axioms are introduced, the α -sensitivity condition of the comorphism does not hold.

Example 7.2.3 This example actually contains two translations, CASL2HasCASL [Mossakowski, 2005] and CASL2Isabelle (translation (7) in [Mossakowski, 2002b]), which are similar in the sense that the same type of problem is encountered when translating theories. Both in Haskell and in Isabelle, it is essential to know the constructors of a datatype when doing the analysis of program blocks, because pattern-matching is allowed only against the constructors. Therefore, this information has to be stored in the signature (see Fig. 7.5 with the translations of the CASL theory NAT from Fig. 7.3 to HasCASL, where 0 and suc are constructors in the resulting HasCASL theory and displayed as such), unlike the case of CASL. This causes the theory mapping of the comorphism not to be signature-preserving, as it depends on the presence of sort generation constraints.

It is not a solution to keep the constructors of datatypes in the CASL signatures

- $(0 + 0 = 0 + 0$
 $\wedge \forall y : \text{Nat} \bullet 0 + y = y + 0 \Rightarrow 0 + \text{succ}(y) = \text{succ}(y) + 0)$
 $\Rightarrow \forall x : \text{Nat} \bullet 0 + x = x + 0$
%(Ax4)%
- $((\forall y : \text{Nat} \bullet \text{succ}(0) + y = 0 + \text{succ}(y))$
 $\wedge \forall y1 : \text{Nat}$
 $\bullet (\forall y : \text{Nat} \bullet \text{succ}(y1) + y = y1 + \text{succ}(y))$
 $\Rightarrow \forall y : \text{Nat}$
 $\bullet \text{succ}(\text{succ}(y1)) + y = \text{succ}(y1) + \text{succ}(y))$
 $\Rightarrow \forall x, y : \text{Nat} \bullet \text{succ}(x) + y = y + \text{succ}(x)$
%(Ax5)%
- $((\forall y : \text{Nat} \bullet 0 + y = y + 0)$
 $\wedge \forall y1 : \text{Nat}$
 $\bullet (\forall y : \text{Nat} \bullet y1 + y = y + y1)$
 $\Rightarrow \forall y : \text{Nat} \bullet \text{succ}(y1) + y = y1 + \text{succ}(y))$
 $\Rightarrow \forall x, y : \text{Nat} \bullet x + y = y + x$
%(Ax6)%

Figure 7.4: Axioms added by the translation CASL2SoftFOLInduction

as well: if we would restrict signature morphisms to map datatypes to datatypes, we would lose many views which are now correct in CASL and if we allow signature morphisms to map datatypes to ordinary sorts, a comorphism from this new logic to CASL which introduces a sort generation constraint axiom for each datatype loses functoriality of the signature translation.

logic HASCASL.PPOLYHOL =

```

spec SPEC =
  type Nat
  op 0 : Nat                                %(constructor)%
  op suc : Nat → Nat                       %(constructor)%
   $\forall X1 : \text{Nat}; Y1 : \text{Nat}$ 
  •  $\text{succ } X1 = \text{succ } Y1 \Leftrightarrow X1 = Y1;$ 
   $\forall Y1 : \text{Nat} \bullet \neg 0 = \text{succ } Y1;$ 
  free type Nat ::= 0 | suc Nat
end

```

Figure 7.5: Translation of specification *Nat* to HasCASL.

7.3 Generalized Comorphisms

The previous examples show that there are cases when logic translations can not be formalized as (theoretical) comorphisms, as one of the two requirements introduced by the α -sensitivity condition (Definition 7.1.1) on the theory mapping components fails to hold. We have therefore to generalize the notion to a concept that does not have these restrictions anymore.

Specification frames [Ehrig et al., 1989] formalize abstract specifications and models of specifications, while there are no notions of sentence and satisfaction.

Definition 7.3.1 A specification frame $\mathcal{F} = (Th, Mod)$ consists of

- a category Th whose objects are called theories and
- a functor $Mod : (Th)^{op} \rightarrow \mathbf{Cat}$ giving the category of models of a theory.

Translation between specification frames provide the generality that they make no restriction on the way the objects of Th are mapped.

Definition 7.3.2 A specification frame comorphism (or representation) $\mu : \mathcal{F} \rightarrow \mathcal{F}'$ consists of

- a functor $\Phi : Th \rightarrow Th'$ and
- a natural transformation $\beta : \Phi^{op}; Mod' \rightarrow Mod$.

For generalizing the concept of theoretical comorphism, we proceed in two steps. First, given an institution, we can associate with it a specification frame.

Definition 7.3.3 [Cornelius et al., 1999] Let $I = (\mathbf{Sig}^I, \mathbf{Sen}^I, \mathbf{Mod}^I, \models^I)$ be an institution. The specification frame associated with I , denoted $SF(I)$, is defined as follows:

- the category of theories Th of $SF(I)$ is $\mathbb{T}h^I$, that is, objects are theories of I and arrows are I -theory morphisms;
- the functor Mod of $SF(I)$ is the functor $\mathbf{Mod}^I : \mathbb{T}h^I \rightarrow \mathbf{Cat}$, assigning to each theory its category of models.

The sentences of the original institution are not completely lost, but rather stored in the theories. We can then define generalized theoretical comorphisms.

Definition 7.3.4 A generalized theoretical institution comorphism $\mu : I \rightarrow I'$ is just a specification frame comorphism $\mu : SF(I) \rightarrow SF(I')$.

Let us denote $genIns$ the category of institutions and generalized theoretical institution comorphisms. We can define a functor, denoted $SF : genIns \rightarrow Spec$, (where $Spec$ is the category of specification frames and specification frame comorphisms) which assigns to each institution its associated specification frame and maps generalized theoretical institution comorphisms identically.

7.4 Heterogeneous Specifications

We now investigate to which extent we can apply the Grothendieck construction (defined in Sec. 3.3) to a graph of institutions and generalized theoroidal institution comorphisms. Thus we can give semantics for heterogeneous specification in our new setting. Generalized comorphisms do not come with an explicit sentence translation component. This means that with the usual definition of Grothendieck institution we can not translate sentences along heterogeneous signature morphisms.

The solution is to add sentences to a specification frame $SF(I)$, using the observation that the sentences of the institution I are not lost, but stored in the theories. Given a theory $T = (\Sigma, E)$ and a Σ -sentence e , we can add e to the set of sentences E to obtain a *theory extension*. This leads us to consider all theory morphisms of source T as sentences of the theory T . We can also define a notion of satisfaction for morphisms, in terms of expansions: a T -model M satisfies a theory morphism $\varphi : T \rightarrow T'$ if there exists at least a φ -expansion of M to a T' -model. The idea of theory extensions as sentences originates from [Mossakowski, 1996].

To make the Grothendieck construction over a diagram $I : Ind^{op} \rightarrow genIns$ of institutions and generalized theoroidal comorphisms, we first compose I with the functor SF to obtain a diagram of specification frames and specification frame comorphisms. We will then investigate the hypotheses under which a specification frame can be extended to an institution, using morphisms as sentences, as described above. Thus, we will have defined a functor INS to $coIns$ and, by further composing $I; SF$ with INS , we get a diagram to $coIns$ for which we can build the known comorphism-based Grothendieck institution.

Definition 7.4.1 Let $Spec^{amalg}$ be the subcategory of $Spec$ such that:

- each object S of $Spec^{amalg}$ has pushouts of theories, with a canonical selection of pushouts such that selected pushouts compose and are weakly semi-exact and
- each morphism of $Spec^{amalg}$ has weak amalgamation property and preserves selected pushouts.

Proposition 7.4.2 Let $S = (Th, Mod)$ be an object of $Spec^{amalg}$. Then $INS(S) = (Sign^I, Sen^I, Mod^I, \models)$, defined as follows:

- $Sign^I = Th$;
- $Mod^I = Mod$;
- for each object T of Th , $Sen^I(T) = Th(T, \bullet)$ ¹;
- for any objects T, T' of Th and any morphism $f \in Th(T, T')$, $Sen^I(f) : Sen^I(T) \rightarrow Sen^I(T')$ is the function that maps each morphism $e : T \rightarrow T_1$

¹ Note that morphisms of source T form rather a class than a set. This problem can be overcome if we consider institutions with small signature categories, which suffice in practical situations.

to the morphism of source T' of the selected pushout of the span formed by e and f .

$$\begin{array}{ccc} T & \xrightarrow{f} & T' \\ \downarrow e & & \downarrow \vartheta_1 = f(e) \\ T_1 & \xrightarrow{\vartheta_2} & T'_1 \end{array}$$

- for any object T of Th , any T -model M and any T -sentence $e : T \rightarrow T'$ in $INS(S)$, $M \models e$ iff there exists a T' -expansion of M .

is an institution.

Proof.

We first prove that Sen^I is functorial.

Let e be a T -sentence and let f, f' be morphisms in Th as in the diagram below:

$$T \xrightarrow{f} T' \xrightarrow{f'} T''$$

Then $Sen^I(f')(Sen^I(f)(e))$ is obtained by successively constructing selected pushouts:

$$\begin{array}{ccccc} T & \xrightarrow{f} & T' & \xrightarrow{f'} & T'' \\ \downarrow e & & \downarrow f(e) & & \downarrow f'(f(e)) \\ T_1 & \xrightarrow{\vartheta_2} & T'_1 & \xrightarrow{\delta_2} & T''_1 \end{array}$$

and $Sen^I(f; f')(e)$ is again obtained via a selected pushout:

$$\begin{array}{ccc} T & \xrightarrow{f; f'} & T'' \\ \downarrow e & & \downarrow f; f'(e) \\ T_1 & \xrightarrow{\eta_2} & T''_2 \end{array}$$

and their equality $f; f'(e) = f'(f(e))$ follows from the requirement that selected pushouts compose to a selected pushout.

The satisfaction condition for $INS(S)$ follows then easily from weakly semi-exactness of S .

□

Proposition 7.4.3 Given two specification frames $S_1 = (Th_1, Mod_1)$ and $S_2 = (Th_2, Mod_2)$ and a specification frame comorphism $\mu = (\Phi, \beta) : S_1 \rightarrow S_2$ in $Spec^{amalg}$, then $\rho = (\Phi, \alpha, \beta) : INS(S_1) \rightarrow INS(S_2)$ (note that the action of ρ on signatures and models is inherited from μ and we only need to define the sentence translation component), where α gives for each object T of Th_1 , the function

$\alpha_T : Sen_1(T) \rightarrow Sen_2(\Phi(T))$ defined by $\alpha_T(e) = \Phi(e)$ for each $e \in Sen_1(T)$:

$$\begin{array}{ccc} T & & \Phi(T) \\ \downarrow e & \mapsto & \downarrow \Phi(e) \\ T' & & \Phi(T') \end{array}$$

is an institution comorphism.

Proof.

The naturality of α is ensured by the fact that translations preserve selected pushouts, while the satisfaction condition of the comorphism follows immediately from weak amalgamation property of μ . \square

Corollary 7.4.4 $INS : Spec^{amalg} \rightarrow coIns$ is a functor.

Note that the hypotheses about the objects and morphisms of $Spec^{amalg}$ have to hold for the institutions in the image of I , for the composition $(I; SF); INS$ to be well defined, as illustrated by the diagram below:

$$Ind^{op} \xrightarrow{I} genIns \xrightarrow{SF} Spec^{amalg} \xrightarrow{INS} coIns$$

Let us compare our resulting institution $(I; SF; INS)^\#$ with the Grothendieck logic obtained by flattening a diagram $D : Ind^{op} \rightarrow coIns$ which only involves institutions existing in the logic graph. The differences are at the levels of sentences and satisfaction relation. In the case of morphisms $\iota : (i, (\Sigma, E)) \rightarrow (i, (\Sigma, E \cup \{e\}))$, where e is a sentence in I^i and ι is the identity morphism, the satisfaction of ι in the Grothendieck institution $I^\#$ coincides with the 'local' satisfaction of the sentence e in institution I^i . From a practical point of view, this is important because it allows us to write specifications using the sentences of the logics and to obtain sentences as morphisms only when translating along a generalized theoroidal comorphism. Moreover, when we translate along a theoroidal comorphism that is α -simple in the sense of [Meseguer, 1989], i.e. the functor φ between theories is the α -extension to theories of a functor taking signatures to theories, then by translation along the corresponding generalized comorphism, we still obtain $\alpha_\Sigma(e)$ as the translation of ι .

7.5 Heterogeneous Proofs

When using generalized theoroidal comorphisms, the sentences of the Grothendieck institution are, as defined in the previous section, theory morphisms of the original institution. We would like to obtain an entailment system on sentences of

$INS(SF(I))$ which extends or at least approximates the entailment system of the original logic I . We start by noticing that the semantic entailment of $INS(SF(I))$ can be expressed in terms of model-theoretic conservativity.

Proposition 7.5.1 *Let I be an institution with pushouts of signatures and weakly semi-exact. Then for any theory T of I , for any set of T -sentences $E \cup \{e\}$ in $INS(SF(I))$, we have $E \models_T e$ in $INS(SF(I))$ if and only if the unique morphism from the colimit of the diagram formed by the theory morphisms in E to the colimit of the diagram formed by the theory morphisms in $E \cup \{e\}$, denoted $\chi_{E,e}$ is model-theoretically conservative.*

Proof: Let us take all sentences in E and let T_E be the colimit of the diagram thus formed; moreover, we denote by a slight abuse $E = e_i; \eta_i$ where e_i is some sentence in E and η_i is the corresponding structural morphism of the colimit. We then make the colimit of the span formed by E and e , as below:

$$\begin{array}{ccc}
 & T & \\
 E \swarrow & & \searrow e \\
 T_E & & T_e \\
 \chi_{E,e} \searrow & & \swarrow \gamma_{E,e} \\
 & T_{E \cup \{e\}} &
 \end{array}$$

We now assume that $\chi_{E,e}$ is model-theoretically conservative and we want to prove that $E \models_T e$. Let M be a T -model such that $M \models E$. By definition this means that there is a T_E -model M_E such that $M_E|_E = M$. Since $\chi_{E,e}$ is model-theoretically conservative, there is a $T_{E \cup \{e\}}$ -model N such that $N|_{\chi_{E,e}} = M_E$. Notice then that $N|_{\gamma_{E,e}}$ is a e -expansion of M , which means $M \models e$.

For the reverse implication, let M_E be a T_E -model and we denote $M = M_E|_E$. Since by hypothesis we have $E \models_T e$, it follows that there is a e -expansion M_e of M . I is weakly semi-exact, so we can amalgamate M_E and M_e to a model N which is the $\chi_{E,e}$ -expansion of M_E and thus $\chi_{E,e}$ is conservative. \square

Unfortunately, there is no known logic-independent characterization or approximation of model-theoretical conservativity based on the proof theoretical one, that could be employed in defining entailment in $INS(SF(I))$. We can instead assume the existence of a "proof-theoretic conservativity" predicate on theory morphisms of I , logic specific, which we denote $PTC(\sigma)$ for a theory morphism σ , with the property that whenever $PTC(\sigma)$ holds, σ is model-theoretically conservative. This would allow us to define entailment in $INS(SF(I))$ based on this predicate:

$$E \vdash e \iff PTC(\chi_{E,e})$$

where $\chi_{E,e}$ is as denoted above. The property of the predicate ensures that entailment thus defined is sound. To prove that the relation \vdash is indeed an entailment

system, one could attempt to make an analysis of the properties that the predicate should fulfill, but it is the case that they can not be completely derived in a logic-independent manner i.e. some of the properties expected for entailment systems rely on the particular choice of the predicate.

In practical situations, proof-theoretical conservativity needs to be studied for the logics involved. For the case of CASL, we refer the reader to [Lüth et al., 2005].

7.6 Conclusions

We have introduced the notion of generalized theoroidal institution comorphism, which eliminates the restrictions on the way theories are translated. This new notion broadens the class of logic encodings formalized as comorphisms. We also describe a framework for heterogeneous specifications based on a graph of institutions and generalized comorphism and give conditions in which we can equip the resulting Grothendieck institution with an entailment system.

Comparing our resulting framework with the comorphism-based Grothendieck institution of [Mossakowski, 2002a], at the heterogeneous specification level the differences are almost invisible for the user, since sentences of the logics can still be used and logic translations that can be formalized as comorphisms do not map sentences in a different way when they are represented as generalized theoroidal comorphisms. Moreover, at this level pushouts of signatures are not actually needed and therefore we can use approximations of colimits i.e. weakly amalgamable cocones, so the hypotheses turn to be equivalent. However, the changes are semantically significant when it comes to heterogeneous proving. The assumption that pushouts should exist is, on one side, mandatory and, on the other side, too strong to hold in all practical situations. In particular, this is also the case for some institutions in the logic graph of HETS.

Future work should concern the study of interaction of logic-specific tools with the heterogeneous sentences, which are now signature morphisms and the implementation of this interaction in HETS.

Acknowledgement. This chapter extends [Codescu, 2009] with full proofs. T. Mossakowski, L. Schröder and D. Lücke provided valuable feedback.

Heterogeneous Colimits and Applications

Contents

8.1	Exactness in Grothendieck Institutions	107
8.2	Example: Heterogeneous Ontologies	109
8.3	Relaxing Colimits and Amalgamation	110
8.4	Algorithms for the Relaxed Setting	112
8.5	Colimits in CASL	116
8.6	Normal Forms of Specifications with Hiding	118
8.7	Conclusion	120

Colimits are a categorical concept, used in particular as a means for combining logical theories and software specifications, see Sec. 3.1.1 for the theory and [Williamson et al., 2001] for a tool computing colimits of specifications that has been successfully used in industrial applications. As we discuss also in Sec. 3.1.1, amalgamation is a property of colimits that allows to put together compatible models of the specifications in the diagram to obtain a model of the colimit specification.

In heterogeneous settings (like the one of HETS), colimits and amalgamation can be obtained under certain conditions [Diaconescu, 2002], but often, these conditions are too strong to be met in practice. Hence, we start from weaker conditions, using both amalgamable colimits as well weakly amalgamable cocones.

We describe an algorithmic method of obtaining weakly amalgamable cocones of heterogeneous diagrams, as an approximation of heterogeneous colimits, better suited for practical situations. We also present the way the graph of logics and logic translations, which is the base for heterogeneous specifications, gets a 2-categorical structure by adding the concept of modification, relating translations that are essentially the same.

8.1 Exactness in Grothendieck Institutions

The following results regarding cocompleteness and exactness of Grothendieck institutions have been proved in [Mossakowski, 2002a].

Theorem 8.1.1 *Let $\mathcal{I}: \text{Ind}^{op} \longrightarrow \text{CoIns}$ be an indexed coinstitution and K be some small category such that*

1. Ind is K -complete (that is, has limits of all diagrams over K),
2. Φ^d is K -cocontinuous for each $d: i \rightarrow j \in Ind$ (meaning that it preserves colimits), and
3. the indexed category of signatures of \mathcal{I} is locally K -cocomplete (the latter meaning that \mathbf{Sig}^i is K -cocomplete for each $i \in |Ind|$).

Then the signature category $\mathbf{Sig}^\#$ of the Grothendieck institution has K -colimits. \square

An indexed coinstitution $\mathcal{I}: Ind^{op} \rightarrow \mathbf{CoIns}$ is called (weakly) locally semi-exact, if each institution I^i is (weakly) semi-exact ($i \in |Ind|$).

\mathcal{I} is called (weakly) semi-exact if for each pullback in Ind

$$\begin{array}{ccc} i & \xleftarrow{d_1} & j1 \\ d_2 \uparrow & & \uparrow e_1 \\ j2 & \xleftarrow{e_2} & k \end{array}$$

the square

$$\begin{array}{ccc} \mathbf{Mod}^i(\Sigma) & \xleftarrow{\beta_\Sigma^{d_1}} & \mathbf{Mod}^{j1}(\Phi^{d_1}(\Sigma)) \\ \beta_\Sigma^{d_2} \uparrow & & \uparrow \beta_\Sigma^{e_1} \\ \mathbf{Mod}^{j2}(\Phi^{d_2}(\Sigma)) & \xleftarrow{\beta_\Sigma^{e_2}} & \mathbf{Mod}^k(\Phi^{e_1}(\Phi^{d_1}(\Sigma))) = \mathbf{Mod}^k(\Phi^{e_2}(\Phi^{d_2}(\Sigma))) \end{array}$$

is a (weak) pullback for each signature Σ in \mathbf{Sig}^i .

Theorem 8.1.2 Assume that the indexed coinstitution $\mathcal{I}: Ind^{op} \rightarrow \mathbf{CoIns}$ fulfills the assumptions of Theorem 8.1.1. Then the Grothendieck institution $\mathcal{I}^\#$ is (weakly) semi-exact if and only if

1. \mathcal{I} is (weakly) locally semi-exact,
2. \mathcal{I} is (weakly) semi-exact, and
3. for all $d: i \rightarrow j \in Ind$, \mathcal{I}^d is (weakly) exact.

The importance of this theorem show up in connection with Prop. 3.2.9:

Corollary 8.1.3 The proof system for development graphs with hiding [Mossakowski et al., 2006a] can be re-used for Grothendieck institutions satisfying the assumptions of Theorem 8.1.2, provided that each of the involved institutions can be mapped (via a model-expansive comorphism) into a proof-supported institution.

8.2 Example: Heterogeneous Ontologies

Example 8.2.1 Let us consider the following formalisation of bibliographical data from [Schorlemmer and Kalfoglou, 2008]: we have a description logic T -Box formalized as an \mathcal{ALC} signature Σ_1 with atomic concepts $B = \{Researcher, Article, Journal\}$ and roles $R = \{name, author, title, hasArticle, impactFactor\}$. The axioms Ax_{DLbib} are

$$Researcher \sqsubseteq \exists name. \top$$

$$Article \sqsubseteq \exists author. \top \sqcap \exists title. \top$$

$$Journal \sqsubseteq \exists name. \top \sqcap \exists hasArticle. \top \sqcap \exists impactFactor. \top$$

This is assumed to be a fragment of a larger \mathcal{ALC} ontology with signature Σ_2 .

On the other hand, there is a similar formalization using the relational schema with signature Σ_3 and axioms Ax_{RelBib} presented in Figure 8.1 as a fragment of a larger relational schema Σ_4 of some relational database.

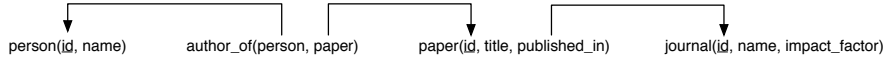


Figure 8.1: Relational schema of an information system

[Schorlemmer and Kalfoglou, 2008] link these two ontologies by mapping both into a given reference ontology T living in first-order logic. However, with this approach, the question whether the axiomatizations Ax_{DLbib} and Ax_{RelBib} have comparable strength cannot be studied at all.

Hence, we here follow a different approach. Instead of using a common reference theory, we specify an interface theory *Interface* in \mathbf{FOL}_{ms} that relates the two ontologies as follows here:

$$\forall p, j, n, f, a, t : s$$

$$\cdot journal(j, n, f) \Leftrightarrow Journal(j) \wedge name(j, n) \wedge impactFactor(j, f)$$

$$\cdot paper(a, t, j) \Leftrightarrow Article(a) \wedge Journal(j) \wedge hasArticle(j, a) \wedge title(a, t)$$

$$\cdot author_of(p, a) \Leftrightarrow Researcher(p) \wedge Article(a) \wedge author(p, a)$$

$$\cdot person(p, n) \Leftrightarrow Researcher(p) \wedge name(p, n)$$

The signature Σ of this interface theory is the union of the translations (as given by the comorphisms from Examples 3.2.4 and 3.2.3) of Σ_1 and Σ_3 to first-order logic.

Assume we want to check whether all models of the theory Ax_{DLbib} in Σ_2 are models of a theory Ax_{RelBib} in Σ_4 . Both theories would have to be translated into a common language. We construct the diagram in figure 8.2 (we marked distinctively the inclusions) and we compute its colimit, then we try to prove that $\vartheta_1(Ax_{DLbib}) \models \vartheta_2(Ax_{RelBib})$, where we denote $\vartheta_1 = \gamma_1; \delta_1$.

□

Using the Heterogeneous Tool Set HETS, we found out that only two of the axioms in Ax_{RelBib} are provable in this way; the second relationship pointing from

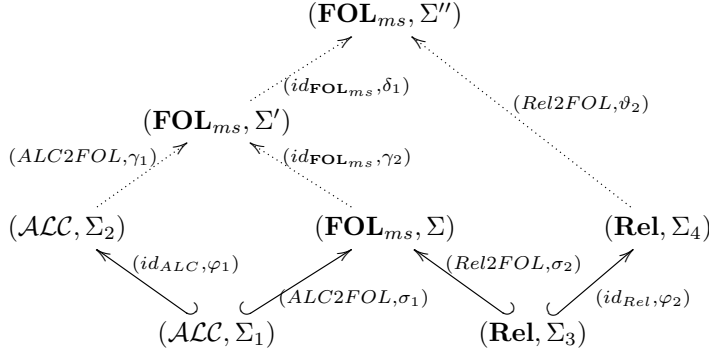


Figure 8.2: A heterogeneous colimit.

the *paper* field of the *author_of* relation to the *id* field of the *paper* relation cannot be deduced from the description logic axiomatization Ax_{DLbib} . The reason for this is that Ax_{DLbib} does not state that an *Article* must appear in *Journal*. In order to extend Ax_{DLbib} accordingly, the inverse of the role *hasArticle* would be needed.

8.3 Relaxing Colimits and Amalgamation

The example of the last section shows that theorems 8.1.1 and 8.1.2 have too strong premises to be applied in all practical situations. Given a diagram $J \rightarrow Ind$, its limit must be the index of some institution that can serve to encode, via comorphisms, all the institutions indexed by the diagram. The existence of such an institution may not be a problem, but the uniqueness condition imposed by the limit property is more problematic. This means that any two such “universal” institutions must have isomorphic indices and hence be isomorphic themselves. This might work well in some circumstances, but may not be desirable in others: after all, a number of non-isomorphic logics, such as classical higher-order logic, the calculus of constructions and rewriting logic have been proposed as such a “universal” logic. Also, the assumptions of Theorem 8.1.2 may not hold in all the cases - e.g. institutions with subsorts [Schröder et al., 2005] or some temporal logics [Martí-Oliet et al., 2004] are not weakly semi-exact.

Therefore, we drop the uniqueness restriction by replacing weak exactness with quasi-exactness, i.e. amalgamable colimits with weakly amalgamable cocones. Thus, the new framework will allow non-exact institutions and comorphisms to be included in the indexed coinstitution serving as basis of the Grothendieck construction.

A problem occurs when using this approach, namely a great number of comorphisms with the same behaviour are introduced via compositions. Therefore, we use the institution comorphism modifications to identify comorphisms with the same sentence and model translation maps. The definitions and results of this section are all from [Mossakowski, 2005].

The idea is to first strengthen the original notion from [Diaconescu, 2002] to

discrete modifications:

Definition 8.3.1 Given institution comorphisms $(\Phi, \rho): I_1 \rightarrow I_2$ and $(\Phi', \rho'): I_1 \rightarrow I_2$, a discrete institution comorphism modification $\vartheta: (\Phi, \rho) \rightarrow (\Phi', \rho')$ is a natural transformation $\vartheta: \Phi \rightarrow \Phi'$ such that $(I_2 \cdot \vartheta) \circ \rho = \rho'$.

Together with obvious identities and compositions, discrete modifications can serve as 2-cells, and thus **CoIns** is turned into a 2-category. Moreover, *Ind* itself is assumed to have a 2-category structure, and the functor \mathcal{I} is then a 2-functor. This leads us to the following definition.

Definition 8.3.2 Given an index 2-category *Ind*, a 2-indexed coinstitution is a 2-functor $\mathcal{I}: \text{Ind}^{op} \rightarrow \text{CoIns}$ into the 2-category of institutions, institution comorphisms and institution comorphism modifications. Note that we may omit the prefix 2- when the 2-category structure is not needed.

We obtain a congruence on Grothendieck signature morphisms: the congruence is generated by

$$(d', \mathcal{I}_\Sigma^u: \Phi^{d'}(\Sigma) \rightarrow \Phi^d(\Sigma)) \equiv (d, id: \Phi^d(\Sigma) \rightarrow \Phi^d(\Sigma))$$

for $\Sigma \in \mathbf{Sig}^i$, $d, d': j \rightarrow i \in \text{Ind}$, and $u: d \Rightarrow d' \in \text{Ind}$. This congruence has the following crucial property:

Proposition 8.3.3 \equiv is contained in the kernel of $\mathcal{I}^\#$ (considered as a functor).

Let $q^\mathcal{I}: \mathbf{Sig}^\# \rightarrow \mathbf{Sig}^\# / \equiv$ be the quotient functor induced by \equiv (see [Mac Lane, 1971] for the definition of quotient category). Note that it is the identity on objects. We easily obtain that the functor $\mathcal{I}^\#$ factors through the quotient category $\mathbf{Sig}^\# / \equiv$.

Corollary 8.3.4 $\mathcal{I}^\#: \mathbf{Sig}^\# \rightarrow \mathbf{Room}$ leads to a quotient Grothendieck institution $\mathcal{I}^\# / \equiv: \mathbf{Sig}^\# / \equiv \rightarrow \mathbf{Room}$.

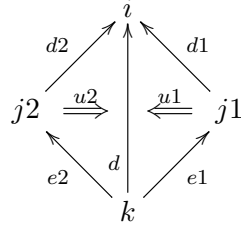
By abuse of notation, we denote $\mathcal{I}^\# / \equiv$ by $(\mathbf{Sig}^\# / \equiv, \mathbf{Sen}^\#, \mathbf{Mod}^\#, \models^\#)$.

Consider the span $\mathbf{PLNG} \xleftarrow{toPFOL^-} \Phi; \mathbf{PFOL}_{ms} \xrightarrow{toPFOL^+} \mathbf{PFOL}_{ms}$ for which we want to obtain a weakly amalgamable cocone. But this can e.g. be given by coding of both \mathbf{PFOL}_{ms} and \mathbf{PLNG} into a common logic such as higher order logic (see Examples 3.2.5 and 3.2.6). However, the resulting square does not commute, since on the way from $\Phi; \mathbf{PFOL}_{ms}$ to \mathbf{HOL}_{ms} via \mathbf{PFOL}_{ms} , the operational semantics of the programming language is expressed in \mathbf{HOL}_{ms} . But there is a diagram of two-cells:

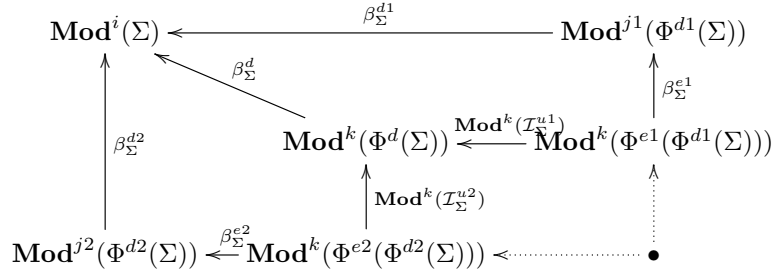
$$\begin{array}{ccccc}
 & & \mathbf{HOL}_{ms} & & \\
 & \nearrow^{PLNG2HOL} & & \nwarrow_{PFOL2HOL} & \\
 \mathbf{PLNG} & \xrightarrow{id} & & \xleftarrow{\vartheta} & \mathbf{PFOL}_{ms} \\
 & \nwarrow_{toPFOL^-} & & \nearrow_{toPFOL^+} & \\
 & & \Phi; \mathbf{PFOL}_{ms} & &
 \end{array}$$

which is weakly amalgamable in the following sense:

Definition 8.3.5 Given a 2-indexed coinstitution $\mathcal{I}: \text{Ind}^{op} \rightarrow \text{CoIns}$, a square consisting of two lax triangles of index morphisms



is called (weakly) amalgamable, if the following diagram is a (weak) pullback for any Σ in Sig^i



where the lower right square is a pullback. That is, each pair consisting of a $\Phi^{d2}(\Sigma)$ - and a $\Phi^{d1}(\Sigma)$ -model with the same Σ -reduct is (weakly) amalgamable to a pair consisting of a $\Phi^{e2}(\Phi^{d2}(\Sigma))$ - and a $\Phi^{e1}(\Phi^{d1}(\Sigma))$ -model having the same $\Phi^d(\Sigma)$ -reduct.

\mathcal{I} is called lax-quasi-exact, if each for pair of arrows $j1 \xrightarrow{d1} i \xleftarrow{d2} j2$ in Ind , there is some weakly amalgamable square of lax triangles as above, such that additionally \mathcal{I}^k is quasi-semi-exact.

Theorem 8.3.6 For a 2-indexed coinstitution $\mathcal{I}: \text{Ind}^* \rightarrow \text{CoIns}$, assume that

- \mathcal{I} is lax-quasi-exact, and
- all institution comorphisms in \mathcal{I} are weakly exact.

Then $\mathcal{I}^\# / \equiv$ is quasi-semi-exact.

8.4 Algorithms for the Relaxed Setting

Call a diagram *connected* if the graph underlying its index category is connected when the identity arrows are deleted. A diagram is *thin*, or a *preorder*, if its index category is thin, i.e. there is at most one arrow between two objects. A preorder is *finitely bounded inf-complete* if any two elements with a common lower bound have an infimum.

Corollary 8.4.1 *Let \mathcal{I} satisfy the assumptions of Theorem 8.3.6. Then $\mathcal{I}^\#/\equiv$ admits weak amalgamation of connected finitely bounded inf-complete diagrams.*

Proof.

Let $D: J \rightarrow \mathbf{Sig}^\#$ be a connected diagram and let Max be the set of maximal nodes in J . We successively construct new diagrams out of J . Take two nodes in Max that have a common lower bound (if two such nodes do not exist, the diagram is not connected). By Theorem 8.3.6, there is a weak amalgamating cocone for the sub-diagram consisting of the two maximal nodes and their infimum (together with the arrows from it into the maximal nodes). Extend the diagram with the cocone. The diagram thus obtained now has a set of maximal nodes whose size is decreased by one. By iterating this construction, we get a diagram with one maximal node. The maximal node then is just the tip of a weakly amalgamating cocone for the original diagram. \square

Analogous to Cor. 8.1.3, we have:

Corollary 8.4.2 *The proof system for development graphs with hiding [Mossakowski et al., 2006a] can be re-used for Grothendieck institutions satisfying the assumptions of Cor. 8.4.1, provided that each of the involved institutions can be mapped (via a model-expansive comorphism) into a proof-supported institution.*

This result leads to a weakly amalgamable square in the Grothendieck institution as follows:

$$\begin{array}{ccc}
 & (\Phi; \mathbf{PFOL}_{ms}, \Sigma_p) & \\
 \begin{array}{c} \swarrow \\ (toPFOL^-, id) \end{array} & & \begin{array}{c} \searrow \\ (toPFOL^+, id) \end{array} \\
 (\mathbf{PLNG}, \Sigma_P) & & (\mathbf{PFOL}_{ms}, \Sigma_S) \\
 \begin{array}{c} \swarrow \\ (PLNG2HOL, id) \end{array} & & \begin{array}{c} \searrow \\ (PFOL2HOL, \vartheta_{\Sigma_S}) \end{array} \\
 & (\mathbf{HOL}_{ms}, PLNG2HOL(\Sigma_P)) &
 \end{array}$$

The algorithm implemented in HETS for obtaining weakly amalgamable cocones of heterogeneous diagrams has some differences with the construction presented in the Corollary 8.4.1. From the practical point of view, it is more convenient not to check whether the entire \mathcal{I} is lax-quasi-exact or if all comorphisms existing in the logic graph are weakly exact, but to test this each time a pair of maximal nodes is chosen. Since the tests may fail to hold for a particular situation, we use backtracking on pairs of maximal nodes and weakly amalgamable squares of lax triangles to explore all possible choices. Also, for homogeneous diagrams, weakly amalgamable cocones are computed within the institution, without further assumptions on the shape of diagram.

Another difference is that the situation when the preorder is not finitely bounded inf-complete is also considered, i.e. the two maximal nodes do not have an infimum, but several maximal common lower bounds. For simplicity, we may

assume that only two such maximal common lower bounds exist, see following diagram:

$$\begin{array}{ccc}
 (i_1, \Sigma_1) & & (i_2, \Sigma_2) \\
 \uparrow (d_1, \varphi_1) & \swarrow & \uparrow (d_4, \varphi_4) \\
 (i_3, \Sigma_3) & \xrightarrow{(d_2, \varphi_2)} & (i_4, \Sigma_4) \\
 & \searrow (d_3, \varphi_3) &
 \end{array}$$

Then, one of these common lower bounds, in our case say (i_3, Σ_3) is selected for building the span for which we compute the weak amalgamating cocone, as in Theorem 8.3.6. Assume this is $(i_1, \Sigma_1) \xrightarrow{(e_1, \rho_1)} (j, \Sigma) \xleftarrow{(e_2, \rho_2)} (i_2, \Sigma_2)$. If the equality $d_3; e_1 = d_4; e_2$ holds and $Sign^j$ has coequalizers, we consider the following double arrow

$$\Phi^{d_3; e_1}(\Sigma_4) \begin{array}{c} \xrightarrow{\Phi^{e_1}(\varphi_3); \rho_1} \\ \xrightarrow{\Phi^{e_2}(\varphi_4); \rho_2} \end{array} \Sigma \xrightarrow{\gamma} \Sigma'$$

for which we compute the coequalizer (γ, Σ') . Then the diagram is extended with $(i_1, \Sigma_1) \xrightarrow{(e_1, \rho_1; \gamma)} (j, \Sigma') \xleftarrow{(e_2, \rho_2; \gamma)} (i_2, \Sigma_2)$. If for a particular choice of maximal bound, one of the assumptions fails to hold, a new common lower bound is selected until all have been considered.

Definition 8.4.3 A weakly amalgamable square of lax triangles

$$\begin{array}{ccccc}
 & & i & & \\
 & d_2 \nearrow & \uparrow & \nwarrow d_1 & \\
 j_2 & \xrightarrow{u_2} & & \xleftarrow{u_1} & j_1 \\
 & \nwarrow e_2 & \downarrow d & \nearrow e_1 & \\
 & & k & &
 \end{array}$$

is sufficiently large for a set of spans in Ind of shape $j_1 \xleftarrow{f_1^a} k_a \xrightarrow{f_2^a} j_2$ if I^k has coequalizers and for each of the spans in the set, $f_1^a; d_1 = f_2^a; d_2$.

A diagram is compatible with squares if for any two maximal nodes of indexes j_1 and j_2 there exists a choice of a maximal common lower bound with the index i and a weakly amalgamable square of lax triangles which is sufficiently large for the set of spans obtained from the other maximal common bounds.

Corollary 8.4.4 Let \mathcal{I} satisfy the assumptions of Theorem 8.3.6. Then $\mathcal{I}^\# / \equiv$ admits weak amalgamation of connected finitely thin diagrams if when extending the diagram with new maximal nodes the compatibility with squares is preserved.

To conclude, the algorithm's steps are the following:

1. Check whether the diagram is homogeneous. If it is, compute a weakly amalgamable cocone in the underlying institution.
2. If the diagram is not connected or not thin, the algorithm fails.
3. Let Max be the set of maximal nodes of the diagram. If Max has only one element, then this is a weakly amalgamable cocone of the original diagram.
4. Pick two maximal nodes that have a common lower bound (the diagram is connected, so we know they exist).
5. Check whether the maximal nodes have an infimum. If they do, compute a weakly amalgamable cocone of the span obtained from the arrows from this infimum to the maximal nodes and extend the diagram with it. Then return to step 3. If we fail to compute the weakly amalgamable cocone (i.e. we do not have the square of lax triangles), return to step 4 to make a new choice.
6. If the two maximal nodes do not have an infimum, compute the list of all maximal common lower bounds of the two nodes.
7. Pick a maximal lower bound and compute a weakly amalgamable cocone of the span obtained from it and the two maximal nodes.
8. For all the others maximal lower bounds, check whether the coequalizers can be computed (as explained above). If this succeeds, extend the diagram with the new node and the arrows to it and go back to step 3. If it fails, return to step 7 to pick another bound. If all the options have failed, return to step 4 to pick new maximal nodes.

The algorithm could find several weakly amalgamable cocones, if there are more squares of lax triangles available for two maximal nodes and a bound. We prefer to display all possible answers and let the user select a cocone, since a certain logic may have better problem-specific tool support. This is also the main advantage over an algorithm that translates the entire diagram to some “universal” logic and computes its colimit.

During the implementation of the algorithm, we also needed to test whether two arbitrary compositions of institution comorphism modifications (as natural transformation, therefore both horizontal and vertical compositions¹, see [Mac Lane, 1971]) are equal. These two kinds of compositions are related by the so-called “Interchange Law” stating that for any natural transformations $\gamma, \mu, \eta, \varepsilon$

$$(\gamma * \eta) \circ (\mu * \varepsilon) = (\gamma \circ \mu) * (\eta \circ \varepsilon)$$

when the compositions on the left side are defined. Using this law and rules for cancelling identities, we develop a term rewrite system (see figure 8.3) which we prove

¹We denote $*$ the horizontal composition and \circ the vertical one.

$(\gamma * \eta) \circ (\mu * \varepsilon) \rightarrow (\gamma \circ \mu) * (\eta \circ \varepsilon)$ $1_F \circ \gamma \rightarrow \gamma$ $\gamma \circ 1_G \rightarrow \gamma$ $1_F * 1_G \rightarrow 1_{F;G}$

Figure 8.3: Rewrite rules for deciding equality of comorphism modifications

terminating and confluent. Then, two arbitrary terms denoting valid compositions are equal if they rewrite to the same normal form.

While termination of the term rewrite system is easy to notice (since for all the rules, on the right side either the depth of term or the number of horizontal compositions decreases), proving confluence is a little more difficult. We used the Church-Rosser checker [Durán and Meseguer, 2010] written in Maude [Clavel et al., 2003] to obtain the critical pairs of the term rewriting system and then noticed that all of them are eliminated in the case of type-correct terms (i.e. modifications that actually compose).

8.5 Colimits in \mathbf{C}_{ASL}

Before discussing applications of colimits in heterogeneous specification and HETS, let us make a couple of remarks about the implementation of homogeneous (i.e. logic-specific) colimits. Of course, colimits of signatures need to be implemented for each of the logics in HETS' graph, as the concept of signature is abstract, with no assumption being made about what an individual signature contains. In most cases, a signature consists of several sets of symbols, e.g. propositions, functions, predicates, sorts, OWL concepts and so on. Colimits of such signatures are then computed in general component-wise, by projecting the graph to one kind of symbols and then computing the colimit in the category of sets, with a well-known construction which we implemented as a generic algorithm. We need to make a convention on how clashing symbols are renamed: to keep generality, we return as result of computing colimits in \mathbf{Set} a set of generic elements paired with the number of the node they originate from and the renaming is handled at the logic-specific level.

However, in the case of \mathbf{C}_{ASL} function and predicate symbols, we must take *overloading* into account: two overloaded symbols in a signature must be named in the same way in the colimit, otherwise, the structural morphism of the signature is not a correct \mathbf{C}_{ASL} morphism. Moreover, we must decide which operation symbol in the colimit must be total. Note that the existence of colimits for diagrams of \mathbf{C}_{ASL} signatures is proven in [Mossakowski, 1998]; here we only present the implementation in HETS and naming decisions.

Example 8.5.1 *Let us consider the following pushout of CASL signatures:*

$$\begin{array}{ccc}
 \begin{array}{l} \mathbf{sorts} \ s, t, u, v, w \\ \mathbf{op} \ d : s \end{array} & \xrightarrow{d \mapsto c} & \begin{array}{l} \mathbf{sorts} \ s, t < u; v, w < z \\ \mathbf{ops} \ c : s; c : t; c : v \end{array} \\
 \downarrow d \mapsto c & & \downarrow c : v \mapsto c_1 : v \\
 \begin{array}{l} \mathbf{sorts} \ s, t < u; v, w; \\ \mathbf{ops} \ c : s; c : t; c : v; c : w \end{array} & \xrightarrow[\begin{array}{l} c : w \mapsto c_1 : w \\ c : v \mapsto c_2 : v \end{array}]{c : w \mapsto c_1 : w} & \begin{array}{l} \mathbf{sorts} \ s, t < u; v, w < z; \\ \mathbf{ops} \ c : s; c : t; \\ c_1 : v; c_2 : v; c_1 : w \end{array}
 \end{array}$$

The sorts of the colimit are obtained by simply projecting the graph of signatures to the corresponding set of sorts and then computing the colimit in **Set**. For the operation symbols, we take again the colimit in **Set** of the sets of operation symbols and then we factor it to overloading classes: the symbols $c : v$ and $c : w$ were not in the overloading relation in the lower left node of the span, but since in the top right node the sorts v and w have a common supersort, they will be overloaded in the colimit. Also, note that in each of the corners of the span we have two constants $c : t$ who do not share a common origin, but they are mapped to symbols in the same overloading class in the colimit, which means that they will not be distinguished. Finally, regarding the choice of names, we renamed the symbol $d : s$ in the source of the span to c as c is the majoritary name for symbols mapped to its overloading class and c_1 and c_2 are generated names.

Let D be a graph of CASL signatures and signature morphisms. The set of function symbols in the colimit signature of D is obtained with the following algorithm:

1. D is projected on operation symbols;
2. We compute colimit of the resulting graph;
3. We factor the resulting set according to the overloading relation in the colimit, determined by the overloading in the nodes of D and the subsort relation in the colimit. A symbol x must be total in the colimit if there is total operation symbol in the graph which is mapped to x . Finally, for each equivalence class thus obtained, we collect all names of the symbols in the graph which are mapped in the equivalence class - we try to keep the original names;
4. For each equivalence class, in order of size - we try to make as few renamings as needed - if the majoritary name does not introduce conflicts, assign it to the symbols, otherwise generate a name using the symbol with the lowest number entering the equivalence class and update the structural morphisms of the colimit.

Finally, we can take advantage of the design of CASL signatures and morphisms: the method computing colimits of CASL signatures (Fig. 8.4) takes as argument a

```

signColimit :: (Category (Sign f e) (Morphism f e m)) =>
  Graph (Sign f e) (Int, Morphism f e m) ->
  (Graph e (Int, m) -> Map Int (Morphism f e m) ->
   (e, Map Int m)) ->
  (Sign f e, Map.Map Int (Morphism f e m))

```

Figure 8.4: Computing colimits of signatures in CASL.

polymorphic method that computes colimits on the extension part of the signature and thus providing colimit computation for a logic that is implemented as a CASL extension reduces to providing such a concrete method on the actual types.

8.6 Normal Forms of Specifications with Hiding

Heterogeneous colimits are needed for several purposes in HETS:

- The proof calculus for development graphs (Sec. 4.3) has been generalised to Grothendieck institutions (Sec. 3.3), see [Mossakowski, 2005]. In order to handle hiding (of parts of specifications) correctly, during a proof, the hidden parts have to be revealed. This is done by computing the so-called *normal forms* of structured specifications using colimits, and in the case of the Grothendieck institutions, these colimits are of course heterogeneous.
- Suppose that we want to prove a view (or refinement) to be correct, which means that we need to show a signature morphism to be a theory morphism. In the development graph, this is represented as a so called global theorem link $\sigma: N_1 \longrightarrow N_2$ (representing a theory morphism) between two nodes N_1 and N_2 (representing the two theories). Note that the logics of N_1 and N_2 may be different and in this case we have to translate N_1 and N_2 into the same logic so we can make the proof. The logical framework approach assumes that the theories of N_1 and N_2 are encoded into some logic that is fixed once and for all. By contrast, in HETS we can rather flexibly find a logic that is a “common upper bound” of the logics of both N_1 and N_2 and that moreover has best possible tool support. This freedom allows us to exploit specialized tools. This is also complemented by a sublogic analysis, which is required for each of the logics in HETS, and which allows for an even more fine-grained determination of available tools. Of course, such “common upper bounds” can be realised as some weak form of colimit.
- Colimits also appear in the semantics of instantiation of parameterised specifications.
- Colimits play a role for alignments of ontologies [Zimmermann et al., 2006], and recently, also heterogeneous ontologies have been studied [Schorlemmer

and Kalfoglou, 2008]. Therefore, we have added a menu for directly computing heterogeneous colimits with HETS.

Here we will only go into details for normal forms of specifications.

For revealing the hidden parts of a structured specification, we can use the construction introduced in [Mossakowski et al., 2001] at the level of development graphs. The idea is that for each node N with incoming hiding definition links we can unfold the subgraph of N into a tree.

Definition 8.6.1 Let $\mathcal{DG} = (\mathcal{N}, \mathcal{E})$ be a development graph and let N be a node of \mathcal{DG} . The diagram $D : I \rightarrow \mathbf{Sig}$ associated with N (where I is a small category which will be defined at the same time with I) is defined inductively together with a map $G : |I| \rightarrow \mathcal{N}$ as follows:

- $\langle N \rangle$ is an object in I such that $D(\langle N \rangle) = \Sigma^N$ ² and $G(\langle N \rangle) = N$;
- if M is an object in I not yet considered:
 - for any incoming global definition link $N \xrightarrow{\sigma} G(M)$ or local definition link $N \xrightarrow{\sigma} G(M)$ in \mathcal{DG} , we add to I a new node M' with $D(M') = \Sigma^N$ and an edge $e : M' \rightarrow M$ with $D(e) = \sigma$ and we set $G(M') = N$;
 - for any incoming hiding definition link $N \xrightarrow[\text{hide}]{\sigma} G(M)$ in \mathcal{DG} , we add to I a new node M' with $D(M') = \Sigma^N$ and an edge $e : M \rightarrow M'$ with $D(e) = \sigma$ and we set $G(M') = N$.

It is possible to have two objects of I mapped to the same node in \mathcal{DG} : if a node is imported into another node via two different paths, the two instances of the imported node should be distinguished.

The normal form of the node N is then defined as the colimit of the diagram D associated with N . D is by construction a connected finitely bounded inf-complete diagram and thus we can compute an approximation of the colimit of D in the heterogeneous case as well. The proof calculus of development graphs uses this normal form node for proofs involving the node N . As heterogeneous colimit computation is now supported by HETS, we have been able to implement as well the corresponding rule of the development graph calculus that deals with hiding.

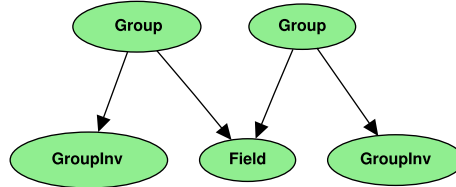
Example 8.6.2 [Mossakowski et al., 2001] Let us consider the following development graph:



where groups are specified first with an inverse operation, which is then hidden. The specification of fields then imports twice the specification **Group**, once for the additive

²Recall that we denoted Σ^N the signature of N in \mathcal{DG} .

and once for the multiplicative operation. Note that appropriate renaming is required to ensure that the symbols of the two laws are not identified. The unfolded subgraph of the node corresponding to the specification of fields and denoted *Field* in the figure above is then as presented below:



and in the normal form of the specification of fields, which is the colimit of this graph, the two inverses are not wrongly identified.

Note that in some cases, unfolding the graph yields too many distinctions between paths: we should consider that a node N is imported into another node M via different paths only if some symbols in Σ^N are mapped to different symbols in Σ^M along the paths. Checking this is computationally expensive, so further investigations are needed before implementing this observation in HETS.

8.7 Conclusion

We have presented an algorithm that generalises the computation of colimits of specifications to a heterogeneous setting. It has turned out that the notion of (amalgamable) colimit has to be replaced by that of weakly amalgamable cocone in order to obtain a framework that is general enough to cover practically interesting cases. Moreover, the algorithm provides a true colimit whenever this is possible. We have illustrated the approach with two examples: one involving the relation between specification and programming. For this example, the 2-categorical machinery is needed in order to construct a weakly amalgamable cocone (which in turn is essential for proving refinements in the proof calculus for development graphs with hiding). The other example concerns ontologies for bibliographical information, and links the schema of a relational database with an ontology specified in description logic. Here, the heterogeneous situation is simpler, because the involved formalisms can be mapped to first-order logic, where also an interface theory lives. However, the logical structure is a bit more complex: in a sense, a refinement between *two* weakly amalgamable cocones needs to be proved. This approach has the advantage (compared with the integration into a common reference ontology pursued in [Schorlemmer and Kalfoglou, 2008]) that the involved axiomatisations can be directly compared w.r.t. their strength. We have pointed out where the strength differs, and how this can be changed if wanted. The integration into a common reference ontology in [Schorlemmer and Kalfoglou, 2008] is of much weaker nature: it just states that the two axiomatisations have a common upper bound. It should be stressed that the cocone computed for this example falls outside the scope

of the standard theorems from the literature [Diaconescu, 2002], because $\mathbf{FOL}_{m,s}$ generally is not the colimit institution for this diagram.

Concerning related work, [Haeusler et al., 2007] tackle the same problem, but involving the invention of new institutions, without making clear how these will be equipped with proof systems. Moreover, amalgamation is not studied at all.

The algorithm is implemented as part of the Heterogeneous Tool Set HETS. Future work should provide more applications to specific examples of heterogeneous specifications and ontologies.

Acknowledgement. This chapter is based on [Codescu and Mossakowski, 2008]. My contribution includes the algorithm in Sec. 8.4, which extends a result from [Mossakowski, 2005] and also the implementation of colimit computation for CASL signatures (Sec. 8.5) and of normal forms of specifications (Sec. 8.6) in HETS.

Integrating Maude into HETS

Contents

9.1	Rewriting Logic and Maude	124
9.2	Relating Maude and CASL Logics	126
9.2.1	Maude Logic	126
9.2.2	Encoding Maude in CASL	127
9.3	From Maude Modules to Development Graphs	128
9.4	Normalization of free definition links	130
9.5	An example: reversing lists	134
9.6	Conclusions and Future Work	136

Maude [Clavel et al., 2007] is a high-level language and high-performance system supporting both equational and rewriting logic computation for a wide range of applications. Maude modules correspond to specifications in rewriting logic, a simple and expressive logic which allows the representation of many models of concurrent and distributed systems.

The key point is that there are three different uses of Maude modules:

1. As programs, to implement some application. We may have chosen Maude because its features make the programming task easier and simpler than other languages.
2. As formal executable specifications, that provide a rigorous mathematical model of an algorithm, a system, a language, or a formalism. Because of the agreement between operational and mathematical semantics, this mathematical model is at the same time executable.
3. As models that can be formally analyzed and verified with respect to different properties expressing various formal requirements. For example, we may want to prove that our Maude module terminates; or that a given function, equationally defined in the module, satisfies some properties expressed as first-order formulas.

However, when we follow this last approach we find that, although Maude can automatically perform analyses like model checking of temporal formulas or verification of invariants, other formal analyses have to be done “by hand,” thus disconnecting the real Maude code from its logical meaning. Although some efforts, like

the Inductive Theorem Prover [Clavel et al., 2006], have been dedicated to palliate this problem, they are restricted to inductive proofs in Church-Rosser equational theories, and they lack the generality to deal with all the features of Maude. With our approach, we cover arbitrary first-order properties (also written in logics different from Maude), and open the door to automated induction strategies such as those of ISApplanner [Dixon and Fleuriot, 2004].

Here we describe an integration of Maude into HETS from which we expect several benefits: On the one hand, Maude will be the first dedicated rewriting engine that is integrated into HETS (so far, only the rewriting engine of Isabelle is integrated, which however is quite specialized towards higher-order proofs). On the other hand, certain features of the Maude module system like views lead to proof obligations that cannot be checked with Maude—HETS will be the suitable framework to prove them, using its above mentioned proof tools.

9.1 Rewriting Logic and Maude

Maude is an efficient tool for equational reasoning and rewriting. Methodologically, Maude specifications are divided into a specification of the data objects and a specification of some concurrent transition system, the states of which are given by the data part. Indeed, at least in specifications with initial semantics, the states can be thought of as equivalence classes of terms. The data part is written in a variant of subsorted conditional equational logic. The transition system is expressed in terms of a binary rewriting relation, and also may be specified using conditional Horn axioms.

Two corresponding logics have been introduced and studied in the literature: rewriting logic and preordered algebra [Meseguer, 1992]. They essentially differ in the treatment of rewrites: whereas in rewriting logic, rewrites are named, and different rewrites between two given states (terms) can be distinguished (which corresponds to equipping each carrier set with a category of rewrites), in preordered algebra, only the existence of a rewrite does matter (which corresponds to equipping each carrier set with a preorder of rewritability).

Rewriting logic has been announced as the logic underlying Maude [Clavel et al., 2007]. Maude modules lead to rewriting logic theories, which can be equipped with loose semantics (*fth/th* modules) or initial/free semantics (*fmod/mod* modules). Although rewriting logic is not given as an institution [Diaconescu, 2008], a specification frame collapsing signatures and sentences into theories would be sufficient for our purposes.

However, after a closer look at Maude and rewriting logic, we found out that *de facto*, the logic underlying Maude differs from the rewriting logic as defined in [Meseguer, 1992]. The reasons are:

1. In Maude, labels of rewrites cannot (and need not) be translated along signature morphisms. This means that *e.g. Maude views do not lead to theory morphisms in rewriting logic!*

2. Although labels of rewrites are used in traces of counterexamples, they play a subsidiary role, because they cannot be used in the linear temporal logic of the Maude model checker.

Specially the first reason completely rules out a rewriting logic-based integration of Maude into HETS: if a view between two modules is specified, HETS definitely needs a theory morphism underlying the view.¹ However, the Maude user does not need to provide the action of the signature morphism on labeled rewrites, and generally, there is more than one possibility to specify this action.

The conclusion is that the most appropriate logic to use for Maude is preordered algebra [Futatsugi and Diaconescu, 1998]. In this logic, rewrites are neither labeled nor distinguished, only their existence is important. This implies that Maude views lead to theory morphisms in the institution of preordered algebras. Moreover, this setting also is in accordance with the above observation that in Maude, rewrite labels are not first-class citizens, but are mere names of sentences that are convenient for decorating tool output (e.g. traces of the model checker). Labels of sentences play a similar role in HETS, which perfectly fits here.

Actually, the switch from rewriting logic to preordered algebras has effects on the consequence relation, contrary to what is said in [Meseguer, 1992].

Example 9.1.1 Consider the following Maude theory:

```
th A is
  sorts S T .
  op a : -> S .
  eq X:S = a .
  ops h k : S -> T .
  rl [r] : a => a .
  rl [s] : h(a) => k(a) .
endfth
```

This logically implies $h(x) \Rightarrow k(x)$ in preordered algebra, but not in rewriting logic, since in the latter logic it is easy to construct models in which the naturality condition $r; k(r) = h(r); s$ fails to hold².

Before describing how to encode Maude into HETS we briefly outline the structuring mechanisms used in Maude specifications:

Module importation. In Maude, a module can be imported in three different modes, each of them stating different semantic constraints: Importing a module in *protecting* mode intuitively means that *no junk and no confusion* are added; importing a module in *extending* mode indicates that junk is allowed, but *confusion is forbidden*; finally, importing a module in *including* mode indicates that *no requirements* are assumed.

¹If the Maude designers would let (and force) users to specify the action of signature morphisms on rewrite labels, it would not be difficult to switch the HETS integration of Maude to being based on rewriting logic.

²This observation is due to T. Mossakowski.

Module summation. The summation module operation creates a new module that includes all the information in its summands.

Renaming. The renaming expression allows to rename sorts, operators (that can be distinguished by their profiles), and labels.

Theories. Theories are used to specify the requirements that the parameters used in parameterized modules must fulfill. Functional theories are membership equational specifications with *loose* semantics. Since the statements specified in theories are not expected to be executed in general, they do not need to satisfy the executability requirements.

Views. A view indicates how a particular module satisfies a theory, by mapping sorts and operations in the theory to those in the target module, in such a way that the induced translations on equations and membership axioms are provable in the module. Note that Maude does not provide a syntax for mapping rewrite rules; however, the existence of rewrites between terms must be preserved by views.

9.2 Relating Maude and CASL Logics

In this section, we will relate Maude and CASL at the level of logical systems. The structuring level will be considered in the next section.

9.2.1 Maude Logic

As already motivated, we will work with preordered algebra semantics for Maude. We will define an institution, that we will denote $Maude^{pre}$, which can be, like in the case of Maude's logic, parametric over the underlying equational logic. Following the Maude implementation, we have used membership equational logic [Meseguer, 1998]. The resulting institution $Maude^{pre}$ is very similar to the one defined in the context of CafeOBJ [Futatsugi and Diaconescu, 1998, Diaconescu, 2008] for pre-ordered algebra (the differences are mainly given by the discussion about operation profiles below, but this is only a matter of representation). This allows us to make use of some results without giving detailed proofs.

Signatures of $Maude^{pre}$ are tuples $(K, F, kind : (S, \leq) \rightarrow K)$, where K is a set (of kinds), $kind$ is a function assigning a kind to each sort in the poset (S, \leq) , and F is a set of function symbols of the form $F = \{F_{k_1 \dots k_n \rightarrow x} \mid k_i, k \in K\} \cup \{F_{s_1 \dots s_n \rightarrow s} \mid s_i, s \in S\}$ such that if $f \in F_{s_1 \dots s_n \rightarrow s}$, there is a symbol $f \in F_{kind(s_1) \dots kind(s_n) \rightarrow kind(s)}$. There is actually no essential difference between our putting operation profiles on sorts into the signatures and Meseguer's original formulation putting them into the sentences.

Given two signatures $\Sigma_i = (K_i, F_i, kind_i)$, $i \in \{1, 2\}$, a signature morphism $\varphi : \Sigma_1 \rightarrow \Sigma_2$ consists of a function $\varphi^{kind} : K_1 \rightarrow K_2$ which preserves \leq_1 , a function

between the sorts $\varphi^{sort} : S_1 \rightarrow S_2$ such that $\varphi^{sort}; kind_2 = kind_1; \varphi^{kind}$ and the subsorts are preserved, and a function $\varphi^{op} : F_1 \rightarrow F_2$ which maps operation symbols compatibly with the types. Moreover, the overloading of symbol names must be preserved, i.e. the name of $\varphi^{op}(\sigma)$ must be the same both when mapping the operation symbol σ on sorts and on kinds. With composition defined component-wise, we get the category of signatures.

For a signature Σ , a model M interprets each kind k as a preorder (M_k, \leq) , each sort s as a subset M_s of $M_{kind(s)}$ that is equipped with the induced preorder, with M_s a subset of $M_{s'}$ if $s < s'$, and each operation symbol $f \in F_{k_1 \dots k_n, k}$ as a function $M_f : M_{k_1} \times \dots \times M_{k_n} \rightarrow M_k$ which has to be monotonic and such that for each function symbol f on sorts, its interpretation must be a restriction of the interpretation of the corresponding function on kinds. For two Σ -models A and B , a homomorphism of models is a family $\{h_k : A_k \rightarrow B_k\}_{k \in K}$ of preorder-preserving functions which is also an algebra homomorphism and such that $h_{kind(s)}(A_s) \subseteq B_s$ for each sort s .

The sentences of a signature Σ are Horn clauses built with three types of atoms: equational atoms $t = t'$, membership atoms $t : s$, and rewrite atoms $t \rightsquigarrow t'$, where t, t' are F -terms and s is a sort in S . Given a Σ -model M , an equational atom $t = t'$ holds in M if $M_t = M_{t'}$, a membership atom $t : s$ holds when M_t is an element of M_s , and a rewrite atom $t \rightsquigarrow t'$ holds when $M_t \leq M_{t'}$. The set of variables X used for quantification is K -sorted. The satisfaction of sentences extends the satisfaction of atoms in the obvious way.

9.2.2 Encoding Maude in CASL

We now present an encoding of Maude into CASL. It can be formalized as a so-called institution comorphism [Goguen and Roşu, 2002]. Note that we will only need the Horn Clause fragment of first-order logic. For freeness (see Sect. 9.4), we will also need sort generation constraints, as well as the *second-order* extension of CASL with quantification over predicates. The idea of the encoding of *Maude^{pre}* in CASL is that we represent rewriting as a binary predicate and we axiomatize it as a preorder compatible with operations.

Every Maude signature $(K, F, kind : (S, \leq) \rightarrow K)$ is translated to the CASL theory $((S', \leq', F, P), E)$, where S' is the disjoint union of K and S , \leq' extends the relation \leq on sorts with pairs $(s, kind(s))$, for each $s \in S$, $rew \in P_{s,s}$ for any $s \in S'$ is a binary predicate and E contains axioms stating that for any kind k , $rew \in P_{k,k}$ is a preorder compatible with the operations. The latter means that for any $f \in F_{s_1 \dots s_n, s}$ and any x_i, y_i of sort $s_i \in S'$, $i = 1, \dots, n$, if $rew(x_i, y_i)$ holds, then $rew(f(x_1, \dots, x_n), f(y_1, \dots, y_n))$ also holds.

Let Σ_i , $i = 1, 2$ be two Maude signatures and let $\varphi : \Sigma_1 \rightarrow \Sigma_2$ be a Maude signature morphism. Then its translation $\Phi(\varphi) : \Phi(\Sigma_1) \rightarrow \Phi(\Sigma_2)$ denoted φ , is defined as follows:

- for each $s \in S$, $\varphi(s) := \varphi^{sort}(s)$ and for each $k \in K$, $\varphi(k) := \varphi^{kind}(k)$.

- the subsort preservation condition of φ follows from the similar condition for φ .
- for each operation symbol σ , $\varphi(\sigma) := \varphi^{op}(\sigma)$.
- *rew* is mapped identically.

The sentence translation map for each signature is obtained in two steps. While the equational atoms are translated as themselves, membership atoms $t : s$ are translated to CASL memberships $t \text{ in } s$ and rewrite atoms of form $t \rightsquigarrow t'$ are translated as $rew(t, t')$. Then, any sentence of Maude of the form $(\forall x_i : k_i)H \implies C$, where H is a conjunction of Maude atoms and C is an atom is translated as $(\forall x_i : k_i)H' \implies C'$, where H' and C' are obtained by mapping all the Maude atoms as described before.

Given a Maude signature Σ , a model M' of its translated theory (Σ', E) is reduced to a Σ -model denoted M where:

- for each kind k , define $M_k = M'_k$ and the preorder relation on M_k is *rew*;
- for each sort s , define M_s to be the image of M'_s under the injection $inj_{s, kind(s)}$ generated by the subsort relation;
- for each f on kinds, let $M_f(x_1, \dots, x_n) = M'_f(x_1, \dots, x_n)$ and for each f on sorts of result sort s , let $M_f(x_1, \dots, x_n) = inj_{s, kind(s)}(M'_f(x_1, \dots, x_n))$. M_f is monotone because axioms ensure that M'_f is compatible with *rew*.

The reduct of model homomorphisms is the expected one; the only thing worth noticing is that $h_{kind(s)}(M_s) \subseteq N_s$ for each sort s follows from the CASL model homomorphism condition of h .

The model reduct is an isomorphism of categories.

9.3 From Maude Modules to Development Graphs

We describe in this section how Maude structuring mechanisms described in Section 9.1 are translated into development graphs. Then, we explain how these development graphs are normalized to deal with freeness constraints.

Signature morphisms are produced in different ways; explicitly, renaming of module expressions and views lead to signature morphisms; however, implicitly we also find other morphisms: the sorts defined in the theories are qualified with the parameter in order to distinguish sorts with the same name that will be instantiated later by different ones; moreover, sorts defined (not imported) in parameterized modules can be parameterized as well, so when the theory is instantiated with a view these sorts are also renamed (e.g. the sort `List{X}` for generic lists can become `List{Nat}`).

Each Maude module generates two nodes in the development graph. The first one contains the theory equipped with the usual loose semantics. The second one,

linked to the first one with a free definition link (whose signature morphism is detailed below), contains the same signature but no local axioms and stands for the free models of the theory. Note that Maude theories only generate one node, since their initial semantics is not used by Maude specifications. When importing a module, we will select the node used depending on the chosen importation mode:

- The `protecting` mode generates a non-persistent free link between the current node and the node standing for the free semantics of the included one.
- The `extending` mode generates a global link with the annotation `PCons?`, that stands for proof-theoretic conservativity and that can be checked with a special conservativity checker that is integrated into HETS.
- The `including` mode generates a global definition link between the current node and the node standing for the loose semantics of the included one.

The summation module expression generates a new node that includes all the information in its summands. Note that this new node can also need a node with its free model if it is imported in `protecting` mode.

The model class of parameterized modules consists of free extensions of the models of their parameters, that are persistent on sorts, but not on kinds. This notion of freeness has been studied in [Bouhoula et al., 2000] under assumptions like existence of top sorts for kinds and sorted variables in formulas; our results hold under similar hypotheses. Thus, we use the same non-persistent free links described for `protecting` importation to link these modules with their corresponding theories. Views do not generate nodes in the development graph but theorem links between the node corresponding to the source theory and the node with the free model of the target. However, Maude views provide a special kind of mapping between terms, that can in general map functions of different arity. When this mapping is used we generate a new inner node extending the signature of the target to include functions of the adequate arity.

We illustrate how to build the development graph with an example. Consider the following Maude specifications:

```
fmod M1 is
  sort S1 .
  op _+_ : S1 S1 -> S1 [comm] .
endfm

th T is
  sort S1 .
  op _._ : S1 S1 -> S1 .
  eq V1:S1 . V2:S1 = V2:S1 . V1:S1
    [nonexec] .
endth

mod M is
  fmod M2 is
    sort S2 .
  endfm
  mod M3{X :: T} is
    sort S4 .
  endm
  view V from T to M is
```

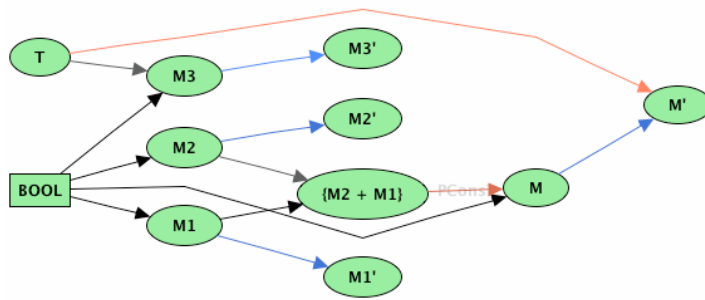


Figure 9.1: Development Graph for Maude Specifications

```

ex M1 + M2 * (sort S2 to S) .          op _._ to _+_ .
endm                                  endv

```

HETS builds the graph shown in Fig. 9.1, where the following steps take place:

- Each module has generated a node with its name and another primed one that contains the initial model, while both of them are linked with a non-persistent free link. Note that theory T did not generate this primed node.
- The summation expression has created a new node that includes the theories of M1 and M2, importing the latter with a renaming; this new node, since it is imported in `extending` mode, uses a link with the `PCons?` annotation.
- There is a theorem link between T and the free (here: initial) model of M. This link is labeled with the mapping defined in the view V.
- The parameterized module M3 includes the theory of its parameter with a renaming, that qualifies the sort. Note that these nodes are connected by means of a non-persistent freeness link.

It is straightforward to show:

Theorem 9.3.1 *The translation of Maude modules into development graphs is semantics-preserving.*

Once the development graph is built, we can apply the (logic independent) calculus rules that reduce global theorem links to local theorem links, which are in turn discharged by local theorem proving. This can be used to prove Maude views, like e.g. “natural numbers are a total order.” We show in the next section how we deal with the freeness constraints imposed by free definition links.

9.4 Normalization of free definition links

Maude uses initial and free semantics intensively. The semantics of freeness is, as mentioned, different from the one used in CASL in that the free extensions of models

are required to be persistent only on sorts and new error elements can be added on the interpretation of kinds. Attempts to design the translation to CASL in such a way that Maude free links would be translated to usual free definition links in CASL have been unsuccessful. We decided thus to introduce a special type of links to represent Maude's freeness in CASL. In order not to break the development graph calculus, we need a way to normalize these links, by replacing them with a semantically equivalent development graph in CASL. The main idea is to make a free extension persistent by duplicating parameter sorts appropriately, such that the parameter is always explicitly included in the free extension.

For any Maude signature Σ , let us define an extension $\Sigma^\# = (S^\#, \leq^\#, F^\#, P^\#)$ of the translation $\Phi(\Sigma)$ of Σ to CASL as follows:

- $S^\#$ unites with the sorts of $\Phi(\Sigma)$ the set $\{[s] \mid s \in \text{Sorts}(\Sigma)\}$;
- $\leq^\#$ extends the subsort relation \leq with pairs $(s, [s])$ for each sort s and $([s], [s'])$ for any sorts $s \leq s'$;
- $F^\#$ adds the function symbols $\{f : [w] \rightarrow [s]\}$ for all function symbols on sorts $f : w \rightarrow s$;³
- $P^\#$ adds the predicate symbol *rew* on all new sorts.

Now, we consider a Maude non-persistent free definition link and let $\sigma : \Sigma \rightarrow \Sigma'$ be the morphism labeling it.⁴ We define a CASL signature morphism $\sigma^\# : \Phi(\Sigma) \rightarrow \Phi(\Sigma')$: on sorts, $\sigma^\#(s) := \sigma^{\text{sort}}(s)$ and $\sigma^\#([s]) := [\sigma^{\text{sort}}(s)]$; on operation symbols, we can define $\sigma^\#(f) := \sigma^{\text{op}}(f)$ and this is correct because the operation symbols were introduced in $\Sigma'^\#$; *rew* is mapped identically.

The normalization of Maude freeness is then illustrated in Fig.9.2. Given a free non-persistent definition link $M \xrightarrow[\text{n.p.free}]{\sigma} N$, with $\sigma : \Sigma \rightarrow \Sigma_N$, we first take the translation of the nodes to CASL (nodes M' and N') and then introduce a new node, K , labeled with $\Sigma_N^\#$, a global definition link from M' to M'' labeled with the inclusion ι_N of Σ_N in $\Sigma_N^\#$, a free definition link from M'' to K labeled with $\sigma^\#$ and a hiding definition link from K to N' labeled with the inclusion $\iota_{N'}$.⁵

The models of N are Maude reducts of CASL models of K , reduced along the inclusion $\iota_{N'}$.

The next step is to eliminate CASL free definition links. The idea is to use a transformation specific to the second-order extension of CASL to normalize freeness. The intuition behind this construction is that it mimics the quotient term algebra

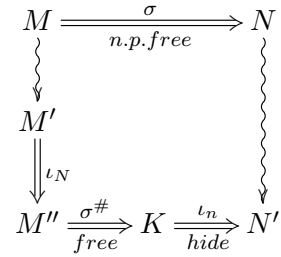


Figure 9.2: Normalization of Maude free links

³ $[x_1 \dots x_n]$ is defined to be $[x_1] \dots [x_n]$.

⁴In Maude, this would usually be an injective renaming.

⁵The arrows without labels in Fig.9.2 correspond to heterogeneous links from Maude to CASL.

construction, that is, the free model is specified as the homomorphic image of an absolutely free model (i.e. term model).

We are going to make use of the following known facts [Reichel, 1987]:

Fact 9.4.1 *Extensions of theories in Horn form admit free extensions of models.*

Fact 9.4.2 *Extensions of theories in Horn form are monomorphic.*

Given a free definition link $M \xrightarrow[\text{free}]{\sigma} N$, with $\sigma : \Sigma \rightarrow \Sigma^N$ such that $Th(M)$ is in Horn form, replace it with $M \xrightarrow{\text{incl}} K \xrightarrow[\text{hide}]{\text{incl}} N'$, where N' has the same signature as N , *incl* denotes inclusions and the node K is constructed as follows.

The signature Σ^K consists of the signature Σ^M disjointly united with a copy of Σ^M , denoted $\iota(\Sigma_M)$ which makes all function symbols total (let us denote $\iota(x)$ the corresponding symbol in this copy for each symbol x from the signature Σ^M) and augmented with new operations $h : \iota(s) \rightarrow ?s$, for any sort s of Σ^M and $make_s : s \rightarrow \iota(s)$, for any sort s of the source signature Σ of the morphism σ labelling the free definition link.

The axioms ψ^K of the node K consist of:

- sentences imposing the bijectivity of *make*;
- axiomatization of the sorts in $\iota(\Sigma_M)$ as free types with all operations as constructors, including *make* for the sorts in $\iota(\Sigma)$;
- homomorphism conditions for h :

$$h(\iota(f)(x_1, \dots, x_n)) = f(h(x_1), \dots, h(x_n))$$

and

$$\iota(p)(t_1, \dots, t_n) \Rightarrow p(h(t_1), \dots, h(t_n))$$

- surjectivity of homomorphisms:

$$\forall y : s. \exists x : \iota(s). h(x) \stackrel{e}{=} y$$

- a second-order formula saying that the kernel of h is the least partial predicative congruence⁶ satisfying $Th(M)$. This is done by quantifying over a predicate symbol for each sort for the binary relation and one predicate symbol for each relation symbol as follows:

⁶A partial predicative congruence consists of a symmetric and transitive binary relation for each sort and a relation of appropriate type for each predicate symbol.

$$\forall \{P_s : \iota(s), \iota(s)\}_{s \in \text{Sorts}(\Sigma_M)}, \{P_{p:w} : \iota(w)\}_{p:w \in \Sigma_M} \\ \cdot \text{symmetry} \wedge \text{transitivity} \wedge \text{congruence} \wedge \text{satThM} \implies \text{largerThenKerH}$$

where *symmetry* stands for

$$\bigwedge_{s \in \text{Sorts}(\Sigma^M)} \forall x : \iota(s), y : \iota(s). P_s(x, y) \implies P_s(y, x),$$

transitivity stands for

$$\bigwedge_{s \in \text{Sorts}(\Sigma^M)} \forall x : \iota(s), y : \iota(s), z : \iota(s). P_s(x, y) \wedge P_s(y, z) \implies P_s(x, z),$$

congruence stands for

$$\bigwedge_{f_{w \rightarrow s} \in \Sigma^M} \forall x_1 \dots x_n : \iota(w), y_1 \dots y_n : \iota(w). \\ D(\iota(f_{w,s})(\bar{x})) \wedge D(\iota(f_{w,s})(\bar{y})) \wedge P_w(\bar{x}, \bar{y}) \implies P_s(\iota(f_{w,s})(\bar{x}), \iota(f_{w,s})(\bar{y}))$$

and

$$\bigwedge_{p_w \in \Sigma^M} \forall x_1 \dots x_n : \iota(w), y_1 \dots y_n : \iota(w). \\ D(\iota(f_{w,s})(\bar{x})) \wedge D(\iota(f_{w,s})(\bar{y})) \wedge P_w(\bar{x}, \bar{y}) \implies P_{p:w}(\bar{x}) \Leftrightarrow P_{p:w}(\bar{y})$$

where D indicates definedness. *satThM* stands for

$$\text{Th}(M) \stackrel{e}{=} /P_s; p : w/P_{p:w}; D(t)/P_s(t, t); t = u/P_s(t, u) \vee (\neg P_s(t, t) \wedge \neg P_s(u, u))$$

where, for a set of formulas Ψ , $\Psi[sy_1/sy'_1; \dots; sy_n/sy'_n]$ denotes the simultaneous substitution of sy'_i for sy_i in all formulas of Ψ (while possibly instantiating the meta-variables t and u). Finally *largerThenKerH* stands for

$$\bigwedge_{s \in \text{Sorts}(\Sigma^M)} \forall x : \iota(s), y : \iota(s). h(x) \stackrel{e}{=} h(y) \implies P_s(x, y) \\ \bigwedge \bigwedge_{p_w \in \Sigma^M} \forall \bar{x} : \iota(w). \iota(p : w)(\bar{x}) \implies P_{p:w}(\bar{x})$$

Proposition 9.4.3 *The models of the nodes N and N' are the same.*

Proof.

Let n be a N -model. To prove that n is also a N' -model, we need to show that it has a K -expansion.

Let us define the following Σ_K model, denoted k :

- on Σ_M , k coincides with n ;
- on $\iota(\Sigma_M)$, the interpretation of sorts and function symbols is given by the free types axioms (i.e. sorts are interpreted as set of terms, operations $\iota(f)$ map terms t_1, \dots, t_n to the term $\iota(f)(t_1, \dots, t_n)$). We define interpretation of predicates after defining h ;

- *make* assigns to each x the term $make(x)$;
- the homomorphism h is defined inductively as follows:
 - $h(make(x)) = x$, if $x \in n_s$ and $s \in Sorts(\Sigma)$;
 - $h(make(t)) = h(t)$, otherwise;
 - $h(\iota(f)(t_1, ..t_n))$ is defined iff $f(h(t_1), ..h(t_n))$ is defined in n and then $h(\iota(f)(t_1, ..t_n)) = f(h(t_1), ..h(t_n))$;
- for predicates in $\iota(\Sigma_M)$ we define $\iota(p)(t_1, ..t_n)$ iff $p(h(t_1), .., h(t_n))$.

The first three types of axioms of the node K hold by construction and $ker(h)$ satisfies $Th(M)$ because n is a M -model. The surjectivity of h and the minimality of $ker(h)$ are exactly the “no junk” and the “no confusion” properties of the free model n .

For the other inclusion, let n' be a model of N' , n_0 be its Σ -reduct and k' a K -expansion of n' . Using the fact that the theory of M is in Horn form, we get an expansion of n_0 to a σ -free model n . We have seen that all free models are also models of N' and moreover we have seen that $ker(k_h)$ is the least predicative congruence satisfying $Th(M)$. The free types axioms of K fix the interpretation of $\iota(\Sigma_M)$ and therefore $ker(k'_h)$ and $ker(k_h)$ are both minimal on the same set, and must be the same. This and the surjectivity of k_h and k'_h allow us to define easily an isomorphism between n and n' and because n' is isomorphic with a free model it must be free as well. \square

9.5 An example: reversing lists

The example we are going to present is a standard specification of lists with empty lists, composition and reversal. We want to prove that by reversing a list twice we obtain the original list. Since Maude syntax does not support marking sentences of a theory as theorems, in Maude we would normally write a view (*PROVEIDEM* in Fig. 9.3, left side) from a theory containing the theorem (*REVIDEM*) to the module with the axioms defining *reverse* (*LISTREV*).

The first advantage the integration of Maude in HETS brings in is that we can use heterogeneous CASL structuring mechanisms and the $\%implies$ annotation to obtain the same development graph in a shorter way – see the right side of Fig. 9.3. We made the convention in HETS to have non-persistent freeness for Maude specifications, modifying thus the usual institution-independent semantics of the freeness construct.

For our example, the order in which the development calculus rules are applied can be summarized as follows. First, the library is translated to CASL; during this step, Maude non-persistent free links are normalized. The next step is to normalize CASL free links, as we defined in the previous section. Note that this introduces hiding definition links and to proceed with proving we first need to compute normal

```

fmod MYLIST is
  sorts Elt List .
  subsort Elt < List .
  op nil : -> List [ctor] .
  op __ : List List -> List
    [ctor assoc id: nil] .
endfm
fmod MYLISTREV is
  pr MYLIST .
  op reverse : List -> List .
  var L : List .
  var E : Elt .
  eq reverse(nil) = nil .
  eq reverse(E L) = reverse(L) E .
endfm
fth REVIDEM is
  pr MYLIST .
  op reverse : List -> List .
  var L : List .
  eq reverse(reverse(L)) = L .
endfth
view PROVEIDEM
  from REVIDEM to MYLISTREV is
  sort List to List .
  op reverse to reverse .
endv

```

```

logic MAUDE
spec PROVEIDEM =
  free
    {sorts Elt List .
     subsort Elt < List .
     op nil : -> List [ctor] .
     op __ : List List -> List
       [ctor assoc id: nil] .
    }
  then {op reverse : List -> List .
        var L : List . var E : Elt .
        eq reverse(nil) = nil .
        eq reverse(E L) = reverse(L) E .
    } then %implies
    {var L : List .
     eq reverse(reverse(L)) = L .
    }

```

Figure 9.3: Lists with reverse, in Maude (left) and CASL (right) syntax.

forms (defined in Section 8.6) for the nodes with incoming hiding links. The rest of the proof follows by applying proof calculus rules in a standard way.

In this way, we now have a proof goal for a second-order theory, introduced while normalizing freeness. It can be discharged using the interactive theorem prover Isabelle/HOL [Nipkow et al., 2002]. We have set up a series of lemmas easing such proofs. First of all, normalization of freeness introduces sorts for the free model which are axiomatized to be the homomorphic image of a set of the absolutely free (i.e. term) model. A transfer lemma (that exploits surjectivity of the homomorphism) enables us to transfer any proof goal from the free model to the absolutely free model. Since the absolutely free model is term generated, we can use induction proofs here. For the case of datatypes with total constructors (like lists), we prove by induction that the homomorphism is total as well. Two further lemmas on lists are proved by induction: (1) associativity of concatenation and (2) the reverse of a concatenation is the concatenation (in reverse order) of the reversed lists. This infrastructure then allows us to demonstrate (again by induction) that $reverse(reverse(L)) = L$.

While proof goals in Horn clause form often can be proved with induction, other proof goals like the inequality of certain terms or extensionality of sets cannot.

Here, we need to prove inequalities or equalities with more complex premises, and this calls for use of the special axiomatization of the kernel of the homomorphism. This axiomatization is rather complex, and we are currently setting up the infrastructure for easing such proofs in Isabelle/HOL.

9.6 Conclusions and Future Work

We have presented how Maude has been integrated into HETS. To achieve this integration, we consider preordered algebra semantics for Maude and define an institution comorphism from Maude to CASL. This integration allows to prove properties of Maude specifications like those expressed in Maude views. We have also implemented a normalization of the development graphs that allows us to prove freeness constraints. We have used this transformation to connect Maude to Isabelle [Nipkow et al., 2002], a Higher Order Logic prover, and have demonstrated a small example proof about reversal of lists. Moreover, this encoding is suited for proofs of e.g. extensionality of sets, which require first-order logic, going beyond the abilities of existing Maude provers like ITP.

Since interactive proofs are often not easy to conduct, future work will make proving more efficient by adopting automated induction strategies like rippling [Dixon and Fleuriot, 2004]. We also have the idea to use the automatic first-order prover SPASS for induction proofs by integrating special induction strategies directly into HETS.

We have also studied the possible comorphisms from CASL to Maude. We distinguish whether the formulas in the source theory are confluent and terminating or not. In the first case, that we plan to check with the Maude termination [Durán et al., 2008] and confluence [Durán and Meseguer, 2010] checkers, we map formulas to equations, whose execution in Maude is more efficient, while in the second case we map formulas to rules.

Finally, we also plan to relate HETS' Modal Logic and Maude models in order to use the Maude model checker [Clavel et al., 2007, Chapter 13] for linear temporal logic.

Acknowledgement. The results of this chapter have appeared in [Codescu et al., 2010b]. I have participated in the implementation of Maude (mainly done by A. Riesco and C. Maeder) and of the translation from Maude to CASL in HETS (Sec. 9.2) and have designed and implemented in HETS the normalization of Maude free definition links (Sec. 9.4) based on a result in [Mossakowski, 2005]. The translation of Maude modules to development graphs (Sec. 9.3) has been done by A. Riesco and C. Maeder.

Part III

Architectural Refinement in HETS

Lambda Expressions in Architectural Specifications

Contents

10.1 Semantics of Generic Unit Expressions	140
10.2 Adding Dependency Tracking	143
10.3 Completeness of Extended Static Semantics	148
10.4 Parametric Architectural Specifications	149
10.5 Refinement of Units with Imports	153
10.6 An Application: Warehouse System	156
10.7 Conclusions	158

The semantics of architectural specifications (Sec. 4.4) relies on compatibility checks between units as prerequisite for combining them. The intuitive idea is that shared symbols must be interpreted in the same way for two models to be put together. The rules have been presented in two ways: the first way is to define a basic static semantics and model semantics in a purely model-theoretical fashion and the compatibility checks are required in the model semantics whenever needed, while the second is an extended static semantics analysis which builds a graph of dependencies between units and discards the compatibility conditions statically. Units of an architectural specification can be *generic* [Sannella and Tarlecki, 1988a], with the intended intuitive meaning that the implementation of the result specification depends on the implementations of the arguments (e.g. some auxiliary functions). Generic units are built using generic unit expressions, written in CASL using the λ -notation: $\lambda X_1 : SP_1, \dots, X_n : SP_n . UT$, where UT is a unit term which contains X_1, \dots, X_n .

The motivation of this chapter is rather technical: the extended static semantics rule for generic unit expression does not keep track of the dependencies between the units used in the unit term UT . This is unsatisfactory for a number of reasons that we give in detail in Section 10.1: firstly, the completeness theorem for extended static semantics (Theorem 5.4 in [Mosses, 2004]) no longer holds when the language is extended with definitions of parametric units. Moreover, *unit imports* are known to be introducing complexity in semantics and verification of architectural specifications. One way to reduce complexity is to replace unit imports with an equivalent construction as below, provided that M is made visible locally in the anonymous architectural specification:

units M : SP1; N : SP2 given M; is equivalent to ...	units M : SP1; N : arch spec { units F : SP1 → SP2 result F[M]}; ...
---	--

If N would be a generic unit, then the result of the architectural specification in the right side would be a λ -expression and the two constructions would no longer be equivalent because they treat differently the dependency between M and N . In Section 10.2 we present our proposed changes for the extended static semantics of architectural specifications, followed by a discussion in Section 10.3 on how the completeness result can be extended to cover lambda expressions as well. Section 10.4 further extends the changes to *parametric* architectural specifications i.e. those having lambda expressions as result. Section 10.5 introduces an extension of the refinement language presented in Chapter 5 to cover refinement of unit with imports, based on the equivalent construction sketched above. Finally, in Section 10.6 we present a larger example motivating the introduction of the new rules, involving refinement of units with imports.

10.1 Semantics of Generic Unit Expressions

We present now the extended static semantic rule for generic unit expressions, with the help of a typical example of a dependency between the unit term of a lambda expression and the generic unit defined by it. Such dependencies are not tracked in the diagram built with the rules for extended static semantics defined in [Mosses, 2004].

Example 10.1.1 *Let us consider the CASL architectural specification below:*

```

spec S = sort s
spec S1 = sort s1
spec S2 = sort s2
arch spec ASP =
units M : S; A1 : S1; A2 : S2;
    L1 =  $\lambda$  X1 : S1 • M and X1;
    L2 =  $\lambda$  X2 : S2 • M and X2;
result L1 [A1] and L2 [A2]
  
```

*The unit term $L1[A1]$ **and** $L2[A2]$ is ill-formed w.r.t. the rules of extended static semantics for architectural specifications because in the diagram in the Fig. 10.1 (built using the extended static semantics rules for generic unit expressions and unit applications, which are presented in Fig. 10.2 and Fig. 10.3 respectively) the sort s can not be traced to a common origin (which should be the node M).*

□

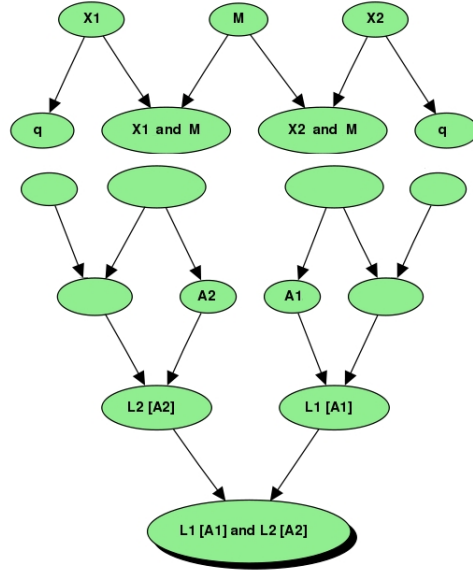


Figure 10.1: Diagram of ASP.

$$\begin{array}{c}
 \Gamma_s \vdash \text{UNIT-BIND-1} \triangleright (UN_1, \Sigma_1) \dots \Gamma_s \vdash \text{UNIT-BIND-n} \triangleright (UN_n, \Sigma_n) \\
 \Sigma_a = \langle \Sigma_1, \dots, \Sigma_n \rangle \text{ and } \Sigma = \Sigma_1 \cup \dots \cup \Sigma_n \\
 UN_1, \dots, UN_n \text{ are new names} \\
 D' \text{ extends } dgm(C_S) \text{ by new node } q \text{ with } D'(q) = \Sigma, \\
 \text{nodes } p_i \text{ and edges } e_i : p_i \rightarrow q \text{ with } D'(e_i) = \iota_{\Sigma_i \subseteq \Sigma} \text{ for } i \in 1, \dots, n \\
 C'_s = (\{\}, \{UN_1 \rightarrow p_1, \dots, UN_n \rightarrow p_n\}, D') \\
 \Gamma_s, C_s + C'_s \vdash \text{UNIT-TERM} \triangleright (p, D'') \\
 D'' \text{ ensures amalgamability along } (D''(p), \langle id_{D''(p)}, \iota_{\Sigma_i \subseteq D''(p)} \rangle_{i \in 1, \dots, n}) \\
 D''' \text{ extends } D'' \text{ by new node } z \text{ with } D'''(z) = \emptyset \\
 \hline
 \Gamma_s, C_s \vdash \text{unit-expr UNIT-BIND-1, } \dots, \text{UNIT-BIND-n UNIT-TERM} \triangleright \\
 (z, \Sigma_a \rightarrow D''(p), D''')
 \end{array}$$

Figure 10.2: Extended static semantics rule for unit expressions (CASL Ref. Manual)

The rule for analysis of generic unit expressions (Fig. 10.2) introduces a node p for the unit term of the lambda expression that keeps track of the sharing information of the terms involved. However, this node p is not further used in application of lambda expressions. In the extended static context, the entry corresponding to the lambda expression only contains a new node labeled with the empty signature, denoted z in Fig. 10.2, as node of imports, and this new node is isolated. Notice also that the purpose of inserting the node q and the edges from nodes p_i to q is to ensure compatibility of the formal parameters when making the analysis of the unit

term.

$$\begin{array}{c}
C_s = (P_s, B_s, D) \\
P_s(UN) = (p^I, (\Sigma_1, \dots, \Sigma_n \rightarrow \Sigma)) \\
\Sigma^F = D(p^I) \cup \Sigma_1 \cup \dots \cup \Sigma_n \\
\Sigma_i, \Gamma_s, C_s \vdash \text{FIT-ARG-i} \triangleright (\sigma_i : \Sigma_i \rightarrow \Sigma_i^A, p_i^A, D_i) \text{ for } i \in 1, \dots, n \\
D_1, \dots, D_n \text{ disjointly extend } D \\
\text{let } D^A = D_1 \cup \dots \cup D_n \\
\Sigma^A = D(p^I) \cup \Sigma_1^A \dots \cup \Sigma_n^A \\
\sigma^A = (id_{D(p^I)} \cup \sigma_1 \cup \dots \cup \sigma_n) : \Sigma^F \rightarrow \Sigma^A \\
\sigma^A(\Delta) : \Sigma \rightarrow (\Sigma^A \cup \Sigma^A(\Delta)), \text{ where } \Delta : \Sigma^F \rightarrow \Sigma \text{ is the signature extension} \\
\Sigma^R = \Sigma^A \cup \Sigma^A(\Delta) \\
D^A \text{ ensures amalgamability along } (\Sigma^A, \langle \iota_{D(p^I) \subseteq \Sigma^A}, \iota_{\Sigma_i^A \subseteq \Sigma^A} \rangle_{i \in 1, \dots, n}) \\
D' \text{ extends } D^A \text{ by new node } q^B, \text{ edge } e^I : p^I \rightarrow q^B \text{ with } D'(e^I) = \iota_{D(p^I \subseteq \Sigma)}, \\
\text{nodes } p_i^F \text{ and edges } e_i^F : p_i^F \rightarrow q^B \text{ with } D'(e_i^F) = \iota_{\Sigma_i \subseteq \Sigma} \\
\text{and } e_i : p_i^F \rightarrow p_i^A \text{ with } D'(e_i) = \sigma_i \text{ for } i \in 1, \dots, n \\
D' \text{ ensures amalgamability along } (\Sigma^R, \langle \sigma^A(\Delta), \iota_{\Sigma_i^A \subseteq \Sigma^R} \rangle_{i \in 1, \dots, n}) \\
D'' \text{ extends } D' \text{ by new node } q, \text{ edge } e' : q^B \rightarrow q \text{ with } D''(e') = \sigma^A(\Delta) \\
\text{and edges } e'_i : p_i^A \rightarrow q \text{ with } D''(e'_i) = \iota_{\Sigma_i^A \subseteq \Sigma^R} \text{ for } i \in 1, \dots, n \\
\hline
\Gamma_s, C_s \vdash \text{unit-appl UN FIT-ARG-1, } \dots, \text{FIT-ARG-n} \triangleright (q, D'')
\end{array}$$

Figure 10.3: Extended static semantics rules for unit application (CASL Ref. Manual)

Using this version of the rules raises a series of problems. First, there is no methodological justification for making terms like the one in our example illegal by not keeping track of the unit M in the lambda expressions. Moreover, the architectural specification ASP has a denotation w.r.t. the basic semantics (it is easy to see that the specification type-checks) and $|ASP|$ has a denotation w.r.t. the model semantics (there is no problem in amalgamating M with a model of specifications $S1$ or $S2$, since there are no shared symbols, and when making the amalgamation of $L1[A1]$ with $L2[A2]$ the symbol s is interpreted in the same way by construction). Thus, since one would expect that the completeness result of [Mosses, 2004] should still hold for the entire architectural language, ASP should have a denotation w.r.t. the extended static semantics.

Another reason to consider the current rules unsatisfactory is the relation between units with imports and generic units. A unit declaration with imports has been informally explained in the literature as a generic unit instantiated once, like in the following example.

Example 10.1.2 *The following unit declarations, taken from the architectural specification of a steam boiler control system (Chapter 13 of [Bidoit and Mosses, 2004]):*

B : BASICS;
 MR : VALUE \rightarrow MESSAGES_RECEIVED *given* B;

can be expressed as a generic unit instantiated once (the linear visibility of units, required in [Mosses, 2004], is assumed to be extended):

B : BASICS;
 MR : **arch spec** {
 units F : BASICS \times VALUE \rightarrow MESSAGES_RECEIVED
 result λ X : VALUE • F [B] [X]};

□

The two declarations in Example 10.1.2 are not equivalent because the former traces the dependency between MR and B while the latter does not. However it has been noticed that to be able to write down refinements of units with imports using the CASL refinement language presented in Chap. 5, this equivalence must become formal. This can only be the case if the second construction also tracks the dependency of B with MR. Another advantage of making the equivalence formal is that the completeness result for extended static semantics and the proof calculus for architectural specifications cover imports as well, since they can now be regarded only as “syntactic sugar” for the equivalent construction.

10.2 Adding Dependency Tracking

The proposed changes are based on the following observation: in the rule for unit application (Fig. 10.3), new nodes are needed for the formal parameters and for the result (labeled p_i^F and q^B respectively). However, for lambda expressions the nodes p_i and p in Fig. 10.2 have already been introduced with the same purpose. This symmetry can be exploited when making the applications of a lambda expression and we will therefore need to keep track of the mentioned nodes.

Recall from [Mosses, 2004] that an extended static unit context consists of a triple (P_s, B_s, D) , where $B_s \in \text{UnitName} \rightarrow \text{Item}$ and stores the corresponding nodes in the diagram for non-generic units, $P_s \in \text{UnitName} \rightarrow \text{Item} \times \text{ParUnitSig}$ and stores the parameterized unit signature of a generic unit together with the node of the imports, such that both B_s and P_s are finite maps and have disjoint domains and D is the signature diagram that stores the dependencies between units.

Firstly, we need to modify the definition of extended static unit contexts such that P_s maps now unit names to pairs in $[\text{Item}] \times \text{ParUnitSig}$, to be able to store the nodes of the parameters and of the result for lambda expressions. A lambda expression must have at least one formal parameter, so the list of items contains either the node of the union of the imports in the case of generic units or at least two elements in the case of definitions of lambda expressions. Moreover, unit declarations of form $UN : \text{arch spec } ASP$ where ASP is an architectural specification whose result unit is a lambda expression also should store the nodes for parameters

$$\begin{array}{c}
\Gamma_s \vdash \text{UNIT-BIND-}i \triangleright (UN_i, \Sigma_i) \text{ for } i \in 1, \dots, n \\
\Sigma_a = \langle \Sigma_1, \dots, \Sigma_n \rangle \text{ and } \Sigma = \Sigma_1 \cup \dots \cup \Sigma_n \\
UN_i \text{ are new names} \\
D' \text{ extends } dgm(C_s) \text{ by new node } q \text{ with } D'(q) = \Sigma, \\
\text{nodes } p_i \text{ with } D'(p_i) = \Sigma_i \\
\text{and edges } e_i : p_i \rightarrow q \text{ with } D'(e_i) = i_{\Sigma_i \subseteq \Sigma} \text{ for } i \in 1, \dots, n \\
C'_s = (\{\}, \{UN_i \rightarrow p_i \mid i \in 1, \dots, n\}, D') \\
\Gamma_s, C_s + C'_s \vdash \text{UNIT-TERM} \triangleright (r, D'') \\
D'' \text{ ensures amalgamability along } (D''(r), \langle id_{D''(r)}, \iota_{\Sigma_i \subseteq D''(r)} \rangle) \\
\cancel{D'''} \text{ extends } D'' \text{ by new node } z \text{ with } D'''(z) = \emptyset \\
\cancel{D'''} \text{ removes from } D'' \text{ the node } q \text{ and its incoming edges} \\
\hline
\Gamma_s, C_s \vdash \text{unit-expr UNIT-BIND-}1, \dots, \text{UNIT-BIND-}n \text{ UNIT-TERM} \triangleright \\
([r, p_1, \dots, p_n], \Sigma_a \rightarrow D'''(r), D''')
\end{array}$$

Figure 10.4: Modified extended static semantics rule for unit expressions.

and the result. The rule changes needed for this latter case are not straightforward and will be addressed separately in Sec. 10.4. In Sec. 10.4 we will also make use of this list of nodes for a different purpose, namely tracking dependencies between different levels of visibility for units.

Fig. 10.4 presents the modified static semantics rule for generic unit expressions, which introduces new nodes p_i for the parameters and a node q to ensure their compatibility during the analysis of the unit term. Then, the result node of the unit term p together with the nodes for parameters are returned as result of the analysis of the lambda expression, together with the diagram resulting by removing the node q and the edges from the nodes p_i to q from the diagram obtained after the analysis of the unit term. The reason why the node q must be removed is that the nodes of the formal parameters will be connected to the actual parameters and their compatibility must be rather checked than ensured. Note that the node q is linked only by the edges added explicitly in D' and therefore no dependency is lost by removing it.

We also have to make a case distinction in the rule of unit application. In the case of generic units, we can use the existing rule for unit applications. The rule for application of lambda expressions is similar with the one used in the first case, but it puts forward the idea that the nodes for formal parameters and result that were stored in the analysis of the lambda expression should be used when making the application. However, this requires special care, as we will illustrate with the help of some examples.

Example 10.2.1 Repeated applications of the same lambda expression. *Let us consider the definition $F = \lambda X : SP . X$ and M where we assume that SP and the specification of M do not share symbols and $M1, M2 : SP$. If we use the stored nodes for parameters and result at every application of F , we obtain the diagram in Fig.*

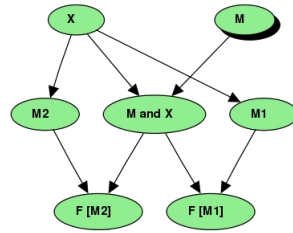


Figure 10.5: Unwanted sharing.

10.5, resulting after applying F to $M1$ and $M2$. The edges from X to $M1$ and $M2$ respectively introduce a sharing requirement between the actual parameters, which is not intended. \square

The solution to this problem is to copy at every application the nodes introduced in the diagram during the analysis of the term of the lambda expression. The copy can be obtained starting with the stored nodes p_i by marking their copies as new formal parameter nodes and going along their outgoing edges: for each new node accessible from p_i , we introduce a copy of it in the diagram together with copies of its incoming edges - this last step copies also the dependencies of the unit term of the lambda expression with the outer units (in the example, the edge from the node of M to the node of M and X is copied). The copying stops when all nodes have been considered, and the copy of the result node is then marked as new result node. Let us denote the procedure described above *copyDiagram*, which takes as inputs the nodes for result and formal parameters of the lambda expression and the current diagram and returns the copied nodes for formal parameters and result and the new diagram. The procedure described works as expected because the diagram created during the analysis of the unit term of the lambda expression consists of exactly the nodes accessible from the formal parameter nodes and it has no cycles; moreover, no new dependencies involving these nodes are ever added in the diagram.

Example 10.2.2 Tracking dependencies of the actual parameters with the environment.

Let us consider the architectural specification below:

```

spec S = sort x
spec T = sorts s, t
spec U = sorts s, u
arch spec ASP =
  units
  P : {sort s};
  A : T given P;
  L =  $\lambda X : S \bullet A$  and X;
  B : U given P

```

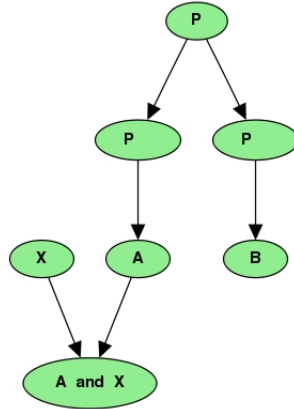


Figure 10.6: Diagram of Example 10.2.2 before application.

result L [B fit $x \mapsto u$]

where the actual parameter and the unit A used in the term of the lambda expression share the sort symbol s , which can be traced in the dependency diagram to a common origin, which is the node of P - see Fig. 10.6. This application should be therefore considered correct. \square

Referring to the rule in Fig. 10.3, the generic unit is given by the inclusion $\Delta : \Sigma^F \rightarrow \Sigma$ of its formal parameters into the body and at application, the fitting arguments give a signature morphism $\sigma^A : \Sigma^F \rightarrow \Sigma^A$ from the formal parameters to the actual parameters. Then, $\Sigma^A \cup \Sigma^A(\Delta)$ results by making the union of the fitting arguments with the body translated along the signature extension $\sigma^A(\Delta) : \Sigma \rightarrow \Sigma^A \cup \Sigma^A(\Delta)$. Originally, an application has been considered not well-formed if the result signature is not a pushout of the body and argument signatures (this is hidden in the use of the notation $\sigma^A(\Delta)$, see [Mosses, 2004]) and this is indeed not the case in Example 10.2.2. We can drop this requirement in the case of lambda expressions and rely on the condition that the diagram should ensure amalgamability; indeed, in this case the application is correct if whenever a symbol is present both in the body and in the argument signatures, the symbol can be traced in the diagram to a common origin which need not be the node of the formal parameter, like in the case of sort s above.

Taking into account the observations in Examples 10.2.1 and 10.2.2, the rule of for application of lambda expressions is presented in Fig. 10.7. Note that we can rely on the fact that a lambda expression has at least one bound variable and therefore the number of nodes stored for parameters is at least one. As we moreover save the node resulting from the analysis of the unit term of the lambda expression, we ensure that a distinction between the application of lambda expressions and of declared generic unit is done and the corresponding rules are used correctly in each case.

$$\begin{array}{c}
C_s = (P_s, B_s, D_0) \\
P_s(UN) = ([p, p_1, \dots, p_n], (\Sigma_1, \dots, \Sigma_n \rightarrow \Sigma)) \\
([r, f_1, \dots, f_n], D) = \text{copyDiagram}([p, p_1, \dots, p_n], D_0) \\
\Sigma^F = \Sigma_1 \cup \dots \cup \Sigma_n \\
\Sigma_i, \Gamma_s, C_s \vdash \text{FIT-ARG-i} \triangleright (\sigma_i : \Sigma_i \rightarrow \Sigma_i^A, p_i^A, D_i) \text{ for } i \in 1, \dots, n \\
D_1, \dots, D_n \text{ disjointly extend } D \\
\text{let } D^A = D_1 \cup \dots \cup D_n \\
\Sigma^A = \Sigma_1^A \cup \dots \cup \Sigma_n^A \\
\sigma^A = (\sigma_1 \cup \dots \cup \sigma_n) : \Sigma^F \rightarrow \Sigma^A \\
\sigma^A(\Delta) : \Sigma \rightarrow (\Sigma^A \cup \Sigma^A(\Delta)), \text{ where } \Delta : \Sigma^F \rightarrow \Sigma \text{ is the signature extension} \\
\text{and the pushout condition for } \Sigma^A \cup \Sigma^A(\Delta) \text{ is dropped} \\
\Sigma^R = \Sigma^A \cup \Sigma^A(\Delta) \\
D^A \text{ ensures amalgamability along } (\Sigma^A, \langle \iota_{\Sigma_i^A \subseteq \Sigma^A} \rangle_{i \in 1, \dots, n}) \\
D' \text{ extends } D^A \text{ with edges } e_i : f_i \rightarrow p_i^A \text{ with } D'(e_i) = \sigma_i, \text{ for } i \in 1, \dots, n \\
D' \text{ ensures amalgamability along } (\Sigma^R, \langle \sigma^A(\Delta), \iota_{\Sigma_i^A \subseteq \Sigma^R} \rangle_{i \in 1, \dots, n}) \\
D'' \text{ extends } D' \text{ by new node } q, \text{ edge } e' : r \rightarrow q \text{ with } D''(e') = \sigma^A(\Delta) \\
\text{and edges } e'_i : p_i^A \rightarrow q \text{ with } D''(e'_i) = \iota_{\Sigma_i^A \subseteq \Sigma^R}, \text{ for } i \in 1, \dots, n \\
\hline
\Gamma_s, C_s \vdash \text{unit-appl UN FIT-ARG-1, \dots, FIT-ARG-n} \triangleright (q, D'')
\end{array}$$

Figure 10.7: Extended static semantics rule for unit application of lambda expressions.

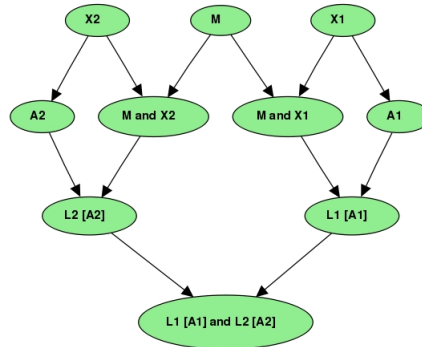


Figure 10.8: Diagram of Example 10.1.1 with the new rules.

Fig. 10.8 presents then the diagram of the architectural specification ASP in Example 10.1.1 using the modified rules of Fig. 10.4 and 10.7¹; in this diagram the sort s can be traced to a common origin and thus the amalgamation is correct. Moreover, when making the application of the lambda expression, the diagram of the term M and X is copied such that no dependency between the actual parameters is incorrectly introduced by edges from the formal parameter node and copying the diagram does not duplicate the node M .

10.3 Completeness of Extended Static Semantics

In this section we discuss how to extend the soundness and completeness result from [Mosses, 2004] to the architectural specification language obtained by adding definitions of generic units to the original fragment language in Section IV.5 of [Mosses, 2004], i.e. unit definitions assign to unit names unit expressions instead of unit terms. Comparing with the language in Fig. 4.2, the differences are that this language does not mix declarations and definitions of units, i.e. all declarations are done locally in the **local** . . . **within** construction, unit declarations do not have imports and unit specifications are never architectural. Also we only restrict to lambda expressions with a single parameter. These differences do not modify the language in an essential way. The soundness and completeness result is formulated as follows.

Theorem 10.3.1 *For any architectural specification ASP in which no generic unit is applied more than once we have that ASP has a denotation w.r.t. the extended static semantics iff ASP has a denotation w.r.t. the static semantics and $|ASP|$ has a denotation w.r.t. the partial model semantics.*

The requirement that no generic unit is applied more than once is a simplifying assumption for achieving a generative semantics².

The theorem is proved using a quite technical lemma (Lemma 5.6 in [Mosses, 2004]) which we do not present in full detail. Intuitively, it says that the extended static semantics for a unit term is successful if and only if the static and model semantics are successful as well and if it is the case, the signatures match and the environment obtained in the model semantics can be represented as a family of models compatible with the diagram obtained in the extended static semantics. The proof of this lemma is done by induction on the structure of the unit term. In order to extend the proof to cover lambda expressions as well, we have two new cases to consider: applications of lambda expressions and local declarations of generic units. The new proof is quite long and tedious, but follows very closely the existing proof. Therefore, we only sketch here the proof idea. For applications of

¹Note that we omitted the nodes of the term of the lambda expression that are copied at each application and only kept the significant ones.

²Recall from Sec. 4.4 that this means that repeated applications of a generic unit to same arguments no longer yields the same result.

lambda expressions, we simply repeat the proof for unit applications but use this time the copies of the nodes for arguments and result that are stored in the context instead of introducing arbitrary distinguished ones. For local declarations of generic units, the proof is similar to the one of local declarations of non-generic units, only that now we have to spell out the rules for lambda expressions before applying the inductive step for the unit term in the lambda expression. The introduced dependency between the lambda expression and its unit term is essential when proving compatibility of the environment with the diagram.

10.4 Parametric Architectural Specifications

Further changes are needed when considering the complete CASL architectural language. The result unit of an architectural specification *ASP* can be itself a lambda expression. In this case the architectural specification is called parametric. The grammar of the architectural language also covers the case when the specification of a unit is itself architectural (either named or anonymous). For such units, we must ensure that designated nodes for formal parameters and result exist in the diagram, since they are required in the rule of unit application of generic units.

Let us first consider the case of anonymous parametric architectural specifications. For the specification below, the static analysis of the architectural specification is currently done in the empty extended static context and thus the nodes for formal parameters and result, which are introduced when making the analysis of the result lambda expression, are no longer present in the diagram at the global level. The dependency between *M* and *F* must be tracked in the diagram in order to ensure correctness of the term $F[M1 \text{ fit } t \mapsto u]$ and $F[M2 \text{ fit } t \mapsto v]$.

```

spec S = sort s
spec T = sort t
spec U = sort u
spec V = sort v
arch spec ASP2 =
  units
  F : arch spec {
    units M : S
    result  $\lambda X : T \bullet M \text{ and } X$ 
  };
  M1 : U; M2 : V;
  result F [M1 fit t  $\mapsto$  u] and F [M2 fit t  $\mapsto$  v]

```

The way we overcome this problem is by making the analysis of the inner architectural specification in the existing global context instead of using an empty global context. After the analysis, we will keep in the global context the diagram resulting from the analysis of the locally-declared units. Thus, the nodes introduced locally become available for further references. Moreover, the units declared locally will

not be kept in the global extended context, since we do not want to extend their scope. By making the analysis of the local specification in the global context, the visibility of units declared at the global level is extended to the local context as well (remember that we assumed this extension of visibility in Example 10.1.2) and the dependencies of the global units with the local environment are tracked by keeping the entire resulting diagram at the global level.

The second case to consider is the one of unit declarations of form $U : \text{arch spec } ASP$, when ASP is a named parametric architectural specification. In this case, ASP cannot refer to units other than those declared within itself and therefore its diagram does not carry any dependency information relevant for the global level. Therefore, instead of adding the diagram of ASP to the global diagram, we only need to introduce new nodes for formal parameters and edges to a new result node. This abstracts away the dependencies of the result node of ASP with the units declared locally (which we do not need) and only keeps the dependencies of the result node with the parameter nodes along the new edges, which will be then copied as diagram of the unit term of the lambda expression at each application of U .

$$\frac{\begin{array}{l} \Gamma_S = (G_s, V_s, A_s, T_s) \\ ASN \text{ is a new name} \\ \Gamma_s, C_0 \vdash \text{ARCH-SPEC} \triangleright (nodes, A\Sigma, D') \end{array}}{\Gamma_S \vdash \text{arch-spec-defn } ASN \text{ ARCH-SPEC} \triangleright (G_s, V_s, A_s \cup \{ASN \mapsto A\Sigma\}, T_s)}$$

Figure 10.9: Rule for architectural library items.

The modifications of the extended static semantics rules are presented in Figures 10.9 to 10.17 and can be summarized as follows. At the library level, the analysis of an architectural specification (Fig. 10.9) starts in the empty extended static unit context. The analysis of an architectural specification (Fig. 10.10), we need to extend the diagram for anonymous parametric architectural specifications (first rule) and named parametric architectural specifications (third rule). In the latter case, we also need to return the (new) nodes for formal parameters and result (r, p_1, \dots, p_n) . The rule for basic architectural specifications (Fig. 10.11) analyzes the list of declarations and definitions in the context received as parameter rather than in the empty context like before. Thus the diagrams built locally will be added to the global diagram and the visibility of global units is extended. The rule for lists of declarations of definitions (Fig. 10.12) modifies the old version just by introducing a unit context C_s^0 in the context of the rule and using it for the analysis of the first declaration instead of C_s^\emptyset . The changes made for unit specifications and result unit expressions (Figures 10.13 to 10.16) are just meant to propagate the results.

The rule for result unit (Fig.10.17) makes a case distinction for each of the four alternatives in Fig. 4.2. When the specification of the unit is not architectural

$$\begin{array}{c}
\boxed{\Gamma_s, C_s \vdash \text{ARCH-SPEC} \triangleright (nodes, A\Sigma, D)} \\
\frac{\Gamma_s, C_s \vdash \text{BASIC-ARCH-SPEC} \triangleright (nodes, A\Sigma, D')}{\Gamma_s, C_s \vdash \text{BASIC-ARCH-SPEC qua ARCH-SPEC} \triangleright (nodes, A\Sigma, D')} \\
\\
\frac{\begin{array}{c} ASN \in Dom(A_s) \\ A_s(ASN) = (S, \Sigma) \\ D' \text{ extends } dgm(C_s) \text{ with a new node } n \text{ such that } D'(n) = \Sigma \end{array}}{(G_s, V_s, A_s, T_s), C_s \vdash ASN \text{ qua ARCH-SPEC} \triangleright ([n], A_s(ASN), dgm(C_s))} \\
\\
\frac{\begin{array}{c} ASN \in Dom(A_s) \\ A_s(ASN) = (S, \langle \Sigma_1, \dots, \Sigma_n \rangle \rightarrow \Sigma) \\ D' \text{ extends } dgm(C_s) \text{ with new nodes } p_1, \dots, p_n, r \text{ and edges } p_i \rightarrow r \\ \text{such that } D'(p_i \rightarrow r) = \iota_{\Sigma_i \subseteq \Sigma} \end{array}}{(G_s, V_s, A_s, T_s), C_s \vdash ASN \text{ qua ARCH-SPEC} \triangleright ([r, p_1, \dots, p_n], A_s(ASN), D')}
\end{array}$$

Figure 10.10: Rules for architectural specifications.

$$\begin{array}{c}
\boxed{\Gamma_s, C_s \vdash \text{BASIC-ARCH-SPEC} \triangleright (nodes, A\Sigma, D)} \\
\frac{\begin{array}{c} \Gamma_s, C_s^0 \vdash UDD^+ \triangleright C_s \\ \Gamma_s, C_s \vdash \text{RESULT-UNIT} \triangleright (nodes, U\Sigma, D) \end{array}}{\Gamma_s, C_s^0 \vdash \text{basic-arch-spec } UDD^+ \text{ RESULT-UNIT} \triangleright (nodes, (ctx(C_s), U\Sigma), D)}
\end{array}$$

Figure 10.11: New extended static semantics rule for basic architectural specifications.

$$\begin{array}{c}
\boxed{\Gamma_s, C_s \vdash \text{UNIT-DECL-DEFN}^+ \triangleright C'_s} \\
\frac{\begin{array}{c} \Gamma_s, C_s^0 \vdash UDD1 \triangleright (C_s)_1 \\ \dots \\ \Gamma_s, (C_s)_{n-1} \vdash UDDn \triangleright (C_s)_n \end{array}}{\Gamma_s, C_s^0 \vdash UDD1, \dots, UDDn \triangleright (C_s)_n}
\end{array}$$

Figure 10.12: New extended static semantics rule for lists of declarations and definitions.

(first two rules), the imported units are analyzed, a new node p labelled with the signature union of all imports is introduced in the diagram and the dependency between the declared unit and the imports is tracked either via the edge from p to q in the first case, or by storing the node p as node of imports in the second case. When the specification of the unit is a parametric architectural specification (third rule), the nodes of formal parameters and results are saved and the unit will be

$$\begin{array}{c}
\boxed{\Gamma_s, C_s \vdash \text{RESULT-UNIT} \triangleright (nodes, U\Sigma, D)} \\
\frac{\Gamma_s, C_s \vdash \text{UNIT-EXPR} \triangleright (p, U\Sigma, D)}{\Gamma_s, C_s \vdash \text{result-unit UNIT-EXPR} \triangleright ([p], U\Sigma, D)} \\
\frac{\Gamma_s, C_s \vdash \text{UNIT-EXPR} \triangleright (r : fs, U\Sigma, D)}{\Gamma_s, C_s \vdash \text{result-unit UNIT-EXPR} \triangleright (r : fs, U\Sigma, D)}
\end{array}$$

Figure 10.13: New extended static semantics rule for result unit expressions.

$$\begin{array}{c}
\boxed{\Gamma_s, C_s \vdash \text{ARCH-UNIT-SPEC} \triangleright (nodes, U\Sigma, D)} \\
\frac{\Gamma_s, C_s \vdash \text{ARCH-SPEC} \triangleright (nodes, (S, U\Sigma), D')}{\Gamma_s, C_s \vdash \text{ARCH-SPEC qua ARCH-UNIT-SPEC} \triangleright (nodes, U\Sigma, D')}
\end{array}$$

Figure 10.14: New extended static semantics rule for architectural unit specifications

$$\begin{array}{c}
\boxed{\Gamma_s, C_s \vdash \text{unit-defn } UN \text{ UNIT-EXPR} \triangleright C'_s} \\
\frac{\Gamma_s, C_s \vdash \text{UNIT-EXPR} \triangleright ([p], \Sigma, D) \quad UN \text{ is a new name}}{\Gamma_s, C_s \vdash \text{unit-defn } UN \text{ UNIT-EXPR} \triangleright (\{\}, \{UN \mapsto (p, \Sigma)\}, D)} \\
\frac{\Gamma_s, C_s \vdash \text{UNIT-EXPR} \triangleright (r : fs, U\Sigma, D) \quad UN \text{ is a new name}}{\Gamma_s, C_s \vdash \text{unit-defn } UN \text{ UNIT-EXPR} \triangleright (\{UN \mapsto (r : fs, U\Sigma)\}, \{\}, D)}
\end{array}$$

Figure 10.15: New rule for unit definitions.

$$\begin{array}{c}
\boxed{\Sigma, \Gamma_s, C_s \vdash \text{UNIT-SPEC} \triangleright (nodes, U\Sigma, D)} \\
\frac{\Gamma_s, C_s \vdash \text{ARCH-UNIT-SPEC} \triangleright (nodes, U\Sigma, D')}{\Sigma, \Gamma_s, C_s \vdash \text{ARCH-UNIT-SPEC qua UNIT-SPEC} \triangleright (nodes, U\Sigma, D')}
\end{array}$$

Figure 10.16: New extended static semantics rule for arch unit specs as unit specs.

applied using the rule for lambda expressions. Finally, when the specification of the unit is a non-parametric architectural specification (last rule), we set the pointer for the unit to the node of the result unit of the architectural specification to be able to trace its dependencies. In the last two cases there are no imports so the node p will always be labeled with the empty signature.

$$\begin{array}{c}
\boxed{\Gamma_s, C_s \vdash \text{UNIT-DECL} \triangleright (C'_s, D)} \\
C_s \vdash \text{UNIT-IMPORTED} \triangleright (p, D) \\
C = C_s + (\{\}, \{\}, D) \\
D(p), \Gamma_s, C \vdash \text{UNIT-SPEC} \triangleright ([], \Sigma, D') \\
UN \text{ is a new name} \\
D'' \text{ extends } D' \text{ by a new node } q \text{ with } D''(q) = D'(p) \cup \Sigma \\
\text{and edge } e : p \rightarrow q \text{ with } D''(e) = \iota_{D'(p) \subseteq D''(q)} \\
\hline
\Gamma_s, C_s \vdash \text{unit-decl } UN \text{ UNIT-SPEC UNIT-IMPORTED} \triangleright (\{\}, \{UN \mapsto q\}, D'') \\
\\
C_s \vdash \text{UNIT-IMPORTED} \triangleright (p, D) \\
C = C_s + (\{\}, \{\}, D) \\
D(p), \Gamma_s, C \vdash \text{UNIT-SPEC} \triangleright ([], \langle \Sigma_1, \dots, \Sigma_n \rangle \rightarrow \Sigma_0, D') \\
UN \text{ is a new name} \\
\hline
\Gamma_s, C_s \vdash \text{unit-decl } UN \text{ UNIT-SPEC UNIT-IMPORTED} \triangleright \\
(\{UN \mapsto (p, \langle \Sigma_1, \dots, \Sigma_n \rangle \rightarrow \Sigma_0 \cup \Sigma^I)\}, \{\}, D') \\
\\
C_s \vdash \text{UNIT-IMPORTED} \triangleright (p, D) \\
C = C_s + (\{\}, \{\}, D) \\
D(p), \Gamma_s \vdash \text{UNIT-SPEC} \triangleright (r : fp, \langle \Sigma_1, \dots, \Sigma_n \rangle \rightarrow \Sigma, D') \\
UN \text{ is a new name} \\
\hline
\Gamma_s, C_s \vdash \text{unit-decl } UN \text{ UNIT-SPEC UNIT-IMPORTED} \triangleright \\
(\{UN \mapsto (r : fp, \langle \Sigma_1, \dots, \Sigma_n \rangle \rightarrow \Sigma)\}, \{\}, D') \\
\\
C_s \vdash \text{UNIT-IMPORTED} \triangleright (p, D) \\
C = C_s + (\{\}, \{\}, D) \\
D(p), \Gamma_s \vdash \text{UNIT-SPEC} \triangleright ([n], \Sigma, D') \\
UN \text{ is a new name} \\
\hline
\Gamma_s, C_s \vdash \text{unit-decl } UN \text{ UNIT-SPEC UNIT-IMPORTED} \triangleright \\
(\{\}, \{UN \mapsto ([n], \Sigma)\}, D')
\end{array}$$

Figure 10.17: New rules for unit declarations.

10.5 Refinement of Units with Imports

In Sec. 10.1 we have discussed that one of the motivations of tracking dependencies between the units used in the unit term of a lambda expression is to simplify the analysis and refinement of units with imports. Recall from Chap. 5 that we introduced units with imports only as syntactic construction, without introducing corresponding rules for their static and model semantics in Sec. 5.2. We complete the definitions in Sec. 5.2 here, and start with a motivating example.

Example 10.5.1 (Refinement of units with imports) *In general, a unit may have several imports, as below:*

arch spec ASP =
 units $M_1 : SP_1$;
 ...
 $M_n : SP_n$;
 $U : SP \rightarrow SP'$ **given** M_1, \dots, M_n ;
 ...
result ...

This can be equivalently expressed using an anonymous architectural specification with a single generic unit which is then applied in the result unit expression to the imported unit terms as below:

arch spec ASP =
 units $M_1 : SP_1$;
 ...
 $M_n : SP_n$;
 $U : \mathbf{arch\ spec} \{$
 units $F : SP_1 \times \dots \times SP_n \times SP \rightarrow SP'$;
 result $\lambda X : SP \bullet F [M_1] \dots [M_n] [X] \}$;
 ...
result ...

The refinement signature of ASP is a branching refinement signature of form $(U\Sigma, BstC)$, with $BstC(U)$ being itself a branching static unit context mapping F to its unit signature.

In general, the imported unit may be written as a unit term of arbitrary complexity instead of just a unit name, like in the example above. In this case, its specification would be no longer directly available. Moreover, as remarked in [Hoffman, 2005], it is not always possible to find a specification that captures exactly the class of all models that may arise as the result of the imported unit term (see also Ex. 11.1.7). It is however possible, as we will see in Chapter 11, to use a proof calculus for architectural specifications similar to the one defined in [Hoffman, 2003] and Section IV.5.3 of [Mosses, 2004] to generate a structured specification that includes this model class among its models. We can then use this structured specification as an approximation.

Note that in Fig. 10.17 the first two rules cover the cases of units with imports. While they suffice for static analysis of architectural specifications in the classical sense, the rules have to be slightly modified when architectural specifications are analyzed as an alternative of refinement specifications. The idea is that the refinement signature of U in the example above must be the same in both equivalent cases. The modification follows easily; we do not present the rules explicitly as

they require many changes to the semantics of architectural specifications to work with refined-unit static contexts instead of unit static context. These changes are however only of formal nature.

We now come to the task of refining units with imports. Given that imports are now only a convenient way for writing down an anonymous architectural specification, we simply need to refine the implicit generic unit introduced in the equivalent syntactic construction, as we illustrate below.

Example 10.5.2 *The refinement signature of ASP from Ex. 10.5.1 is a branching refinement signature $(U\Sigma, BstC)$, with $BstC(UN)$ being itself a branching static unit context mapping F to its unit signature. Notice that when using the first syntactic construction, the unit name F is not in the scope and therefore we can not refine the component UN of ASP as:*

refinement $R = \text{arch spec ASP then } \{UN \text{ to } \{F \text{ to } R'\}\}$

However, the construction of the anonymous architectural specification of UN ensures that it will always have only one unit and therefore we want to allow to write:

refinement $R = \text{arch spec ASP then } \{UN \text{ to } R'\}$

when the signature of UN is a branching static unit context with a single unit name in the domain and the signature of that unit name matches the source signature of R' . This would require that in the composition of refinement signatures, we simply transform the refinement signature $R\Sigma'$ of R' into a component refinement signature mapping the name of the generated generic unit (in our case F) to $R\Sigma'$ before updating the signature of UN in $BstC$ and thus the name F is not made visible to the user. The model semantics rule is similar.

We therefore extend the composition of refinement signatures (Def. 5.2.2) with a new case:

- $R\Sigma_1 = (U\Sigma, BstC)$ with only one unit name UN in the domain of $BstC$ and the composition $R\Sigma'$ of $BstC(UN)$ with $R\Sigma_2$ is defined. In this case, $R\Sigma_1; R\Sigma_2 = (U\Sigma, BstC[\{UN \mapsto R\Sigma'\}])$.

Similarly, the composition of refinement relations (Def. 5.2.6) must be extended as well with a new case:

- $R\Sigma_1 = (U\Sigma, BstC')$ and $BstC'$ has only UN in the domain, and $R\Sigma' = BstC'(UN); R\Sigma_2$ is defined. In this case, $\mathcal{R}_1; \mathcal{R}_2$ is defined as $\mathcal{R}_1; \{UN \mapsto \mathcal{R}_2\}$

This does not introduce any ambiguities between units written at different nesting levels in the same architectural specification, as they have different refinement signatures: if we have a unit declaration of the form $UN : \text{arch spec } \{\text{units } UN : SP; \dots\}$, the signature of the inner declaration of UN is a simple refinement signature, while the signature of the outer declaration of UN is a branching refinement signature.

10.6 An Application: Warehouse System

This section illustrates the use of the new semantics rules for architectural specifications with the help of a case study example - the specification of a warehouse system from [Baumeister and Bert, 2000]. Note that we will also use the CASL refinement language described in Chap. 5 to record formally the refinements introduced in the cited paper only informally, at the meta-level. The system keeps track of stocks of products and of orders and allows adding, canceling and invoicing orders, as well as adding products to the stock.

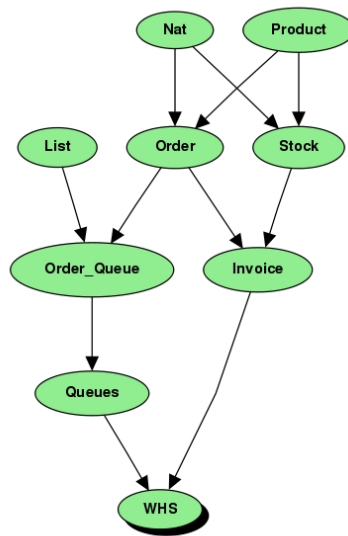


Figure 10.18: Structure of the specification of the warehouse system.

Fig. 10.18 presents the specifications involved and the relations between them. The specifications ORDER, PRODUCT and STOCK specify the objects of the system. The main purpose for the INVOICE specification is to specify an operation for invoicing an order for a product in the stock. The QUEUES and ORDER_QUEUES specifications specify different types of queues (pending, invoiced) for orders. The WHS specification is the top-level specification, with the main operations of the system. The next step is to come up with a more concrete realization of ORDER, that allows to distinguish between different orders on the same quantity of a product by introducing labels. This results in specifications ORDER', INVOICE' and WHS'. The specification WHS' of the warehouse system is then further refined to an architectural specification describing the structure of the implementation of the system. Moreover, NAT and LIST are the usual specifications of natural numbers and lists.

The modular decomposition of the warehouse system is then recorded in the architectural specification below:

arch spec WAREHOUSE =

```

units NATALG : NAT; PRODUCTALG : PRODUCT;
ORDERFUN : PRODUCT → ORDER' given NATALG;
ORDERALG = ORDERFUN [PRODUCTALG];
STOCKFUN : PRODUCT → STOCK given NATALG;
STOCKALG = STOCKFUN [PRODUCTALG];
INVOICEFUN : {ORDER' and STOCK} → INVOICE';
QUEUESFUN : ORDER → QUEUES;
WHSFUN : {QUEUES and INVOICE'} → WHS'
result WHSFUN[QUEUESFUN [ORDERALG]
           and INVOICEFUN [ORDERALG and STOCKALG]]

```

We can write this refinement chain in the following way:

```

refinement R = WHS refined to WHS'
           refined to arch spec WAREHOUSE

```

We can further proceed to refine each component separately. For example, let us assume we want to further refine ORDER' in such a way that the labels of orders are natural numbers and denote the corresponding specification ORDER''.

The changes in the extended static semantics rules allow us to rephrase the declaration of ORDERFUN in an equivalent way using generic units:

```

ORDERFUN : arch spec {
           units F : NAT × PRODUCT → ORDER'
           result lambda X : PRODUCT • F [NATALG] [X] };

```

Then we need to write a unit specification for the specification of ORDERFUN to be able to further refine it:

```

unit spec NATORDER' = NAT × PRODUCT → ORDER'

```

and another unit specification to store the signature after refinement as well:

```

unit spec NATORDER'' = NAT × PRODUCT → ORDER''

```

The refinement is done along a morphism that maps the sort *Label* to *Nat*:

```

refinement R' = NATORDER' refined via Label ↦ Nat to NATORDER''

```

The changes to the semantics of CASL refinement language introduced in the previous section allow us to refine the component ORDERFUN without making use of the arbitrary name (in our case F) chosen for the generic unit:

```

refinement R'' = R then {ORDERFUN to R'}

```

10.7 Conclusions

We have presented and discussed a series of changes to extended static semantics of CASL architectural specifications, motivated by the unsatisfactory treatment of lambda expressions in the original semantics of CASL [Mosses, 2004]. We have identified a number of practically important situations requiring lambda expressions to have dependency tracking with their unit term and we formulated the modified rules accordingly. We have also discussed briefly how the known completeness result can now be successfully extended to the whole CASL architectural language; a full proof is very lengthy and follows the lines of the existing result; for this reason we have omitted it. Finally, we have presented an example of refinement of generic units with imports; without the changes introduced in this paper such a refinement could not have been expressed using the CASL refinement language. Concerning future work, it appears natural to attempt to generalize generic units to the higher-order case. Note that HETS now supports the new rules introduced in this chapter.

Acknowledgement. The results in this chapter have been published in [Code-scu, 2011] and have greatly benefited from suggestions of T. Mossakowski and A. Tarlecki.

Correctness and Consistency of C_{ASL} Refinements

Contents

11.1 Proof Calculus for CASL Architectural Specifications	160
11.2 Proof Calculus for CASL Refinements	171
11.3 Completeness of the Proof Calculus for Refinements	178
11.4 Refinement Trees	184
11.5 Checking Consistency of Refinement Specifications	188
11.6 Conclusion and Future Work	191

In this chapter we look to answer the question whether a refinement specification is correct and consistent. In the most basic form, a refinement is correct when each model of the target specification is mapped by the constructor associated with the implementation step to a model of the source specification of the refinement. The complexity increases in the context of the refinement language of CASL, as it supports branchings and compositions of refinements.

We propose an extension of the proof calculus for the CASL architectural language introduced in [Mosses, 2004] to the entire CASL refinement language. The key idea behind this extension is that given a refinement specification, we construct a structured specification whose denotation is, under conditions we discuss below, exactly the image of the constructor associated with the refinement. Correctness of refinements reduces then to checking model class inclusions. We prove that the calculus thus obtained is sound and moreover complete for the architectural fragment of the CASL refinement language. When considering the whole refinement language however, completeness can only be obtained when exploiting during the verification process information regarding the choice of a particular implementation. We furthermore complement the refinement language with a formal notion of *refinement trees*, which can be thought of as playing a similar role as development graphs do for structured specifications. This means that we can use refinement trees not only to visualize the structure of the development, but also to inspect the involved specifications and to check logical properties of refinements. In particular, we exploit that constructors preserve consistency of specifications to derive a consistency calculus for refinement specifications, which we then connect to the refinement trees in HETS.

$\begin{aligned} \text{ASP} &::= \mathcal{S} \mid \mathbf{units} \text{ UDD}_1 \dots \text{UDD}_n \mathbf{result} \text{ UE} \\ \text{UDD} &::= \text{UDEFN} \mid \text{UDECL} \\ \text{UDECL} &::= \text{UN} : \text{USP} \langle \mathbf{given} \text{ UT}_1, \dots, \text{UT}_n \rangle \\ \text{USP} &::= \text{SP} \mid \text{SP}_1 \times \dots \times \text{SP}_n \rightarrow \text{SP} \mid \mathbf{arch-spec} \text{ ASP} \\ \text{UDEFN} &::= A = \text{UT} \text{ UE} \\ \text{UE} &::= \text{UT} \mid \lambda A_1 : \text{SP}_1, \dots, A_n : \text{SP}_n \bullet \text{UT} \\ \text{UT} &::= A \mid A [\text{FIT}_1] \dots [\text{FIT}_n] \mid \text{UT} \mathbf{and} \text{ UT} \mid \text{UT} \mathbf{with} \sigma : \Sigma \rightarrow \Sigma' \mid \\ &\quad \text{UT} \mathbf{hide} \sigma : \Sigma \rightarrow \Sigma' \mid \mathbf{local} \text{ UDEFN}_1 \dots \text{UDEFN}_n \mathbf{within} \text{ UT} \\ \text{FIT} &::= \text{UT} \mid \text{UT} \mathbf{fit} \sigma : \Sigma \rightarrow \Sigma' \end{aligned}$

Figure 11.1: Restricted CASL architectural language as in [Mosses, 2004]. Our version covers the entire language, with no restrictions.

11.1 Proof Calculus for CASL Architectural Specifications

The main motivation for formalizing the development process using CASL architectural specifications and refinements is that one can then formally *prove* correctness of the entire development. In [Mosses, 2004], Section IV:5, a proof calculus for verification of architectural specifications was introduced as an algorithm for checking whether the resulting units of an architectural specification satisfy a given unit specification. In order to simplify presentation, in [Mosses, 2004] the architectural language was restricted as in Fig. 11.1; however, it is rather straightforward to extend the proof calculus to the whole architectural language, with the notable exception of unit imports. In the following, we will present a new calculus for correctness of architectural specifications, which provides the advantage of being easier to generalize to the whole architectural language, including unit imports. As we have seen in Sec. 10.5, unit imports can be equivalently formulated using an anonymous architectural specification and giving a specification that approximates the model class of each imported unit term. Making use of Dfn. 11.1.6, that introduces such a specification, we can regard unit imports as a syntactic sugar for this construction. Therefore, in the following we assume that unit imports are omitted from the architectural language, since introducing imports will not imply any new conditions on the soundness and completeness of the proof calculus.

We first recall the existing calculus of [Mosses, 2004] and begin by introducing a number of auxiliary concepts. A *context* Γ is a diagram in the signature category of I , whose nodes are additionally labeled with sets of specifications. We will write $A :_{\Sigma} \mathcal{SP}$ to denote that a node A of a context is labeled with the signature Σ and the set of specifications \mathcal{SP} and $\sigma : A \rightarrow B$ to denote an edge between the nodes A and B labeled with σ . Thus, we can regard contexts as sets of such declarations of labeled nodes and edges. We moreover use $A : \Sigma$ to denote that the signature of the node A is Σ . Given a context Γ and a family of models $\mathcal{M} = \{M_p\}_{p \in \text{Nodes}(\Gamma)}$ indexed by the nodes of Γ , we say that \mathcal{M} is *compatible with* Γ if $M_A \in \mathbf{Mod}(\mathcal{SP})$, for each specification \mathcal{SP} in the set labeling the node A in Γ and $M_A = M_B|_{\Gamma(e)}$ for each edge $\sigma : A \rightarrow B$. A *generic context* Γ_{gen} is a finite set of declarations of the

form $A :_{\Sigma \rightarrow \Sigma'} SP \rightarrow SP'$, where $\Sigma \rightarrow \Sigma'$ is the unit signature of $SP \rightarrow SP'$.

The proof calculus of [Mosses, 2004] is then given as below, together with the general format of the rules (in the box in front of them). Its judgements are of the form $\vdash ASP :: USP$, where ASP is an architectural specification over an institution I and USP is a given unit specification. The proof calculus can be regarded as having two components. The first one is a *constructive* component, building a context Γ for keeping track of the dependencies between units as well a generic context Γ_{gen} for storing the generic units. This is done with the rules for unit declarations $UDECL$ and unit terms UT in the proof calculus below. The second component is *deductive* and it uses the contexts Γ and Γ_{gen} built with the constructive component to check whether models of a unit expression satisfy a given unit specification USP . This component contains the rules for architectural specifications ASP and unit expression UE in the proof calculus below. For simplicity, we have chosen to express these proof obligations only semantically; [Mosses, 2004] provides means of expressing them syntactically with the help of the specifications assigned to the nodes in Γ and of discharging them.

$$\boxed{\begin{array}{c} \vdash ASP :: USP \\ \text{(deductive)} \end{array}}$$

$$\begin{array}{c} \vdash UDECL_1 :: \Gamma_{gen}^1, \Gamma^1 \\ \vdots \\ \vdash UDECL_n :: \Gamma_{gen}^n, \Gamma^n \\ \hline \bigcup_{i=1, \dots, n} \Gamma_{gen}^i, \bigcup_{i=1, \dots, n} \Gamma^i \vdash UE :: USP \\ \vdash \mathbf{units} UDECL_1 \dots UDECL_n \mathbf{result} UE :: USP \end{array}$$

$$\boxed{\begin{array}{c} \vdash UDECL :: \Gamma_{gen}, \Gamma \\ \text{(constructive)} \end{array}}$$

$$\vdash A : SP :: \emptyset, \{A :_{\mathbf{Sig}[SP]} SP\}$$

$$\vdash A : SP \rightarrow SP' :: \{A :_{\mathbf{Sig}[SP] \rightarrow \mathbf{Sig}[SP']} SP \rightarrow SP'\}, \emptyset$$

$$\boxed{\begin{array}{c} \Gamma_{gen}, \Gamma \vdash UE :: USP \\ \text{(deductive)} \end{array}}$$

$$\frac{\Gamma_{gen}, \Gamma \vdash UT :: \Gamma', A \text{ for any family of models } \mathcal{M} \text{ compatible with } \Gamma', M_A \in \mathbf{Mod}(SP)}{\Gamma_{gen}, \Gamma \vdash UT \text{ qua } UE :: SP}$$

$$\frac{\begin{array}{l} s\mathbf{Sig}[SP_1] = \mathbf{Sig}[SP] = \Sigma \\ SP \text{ and } SP_1 \text{ are equivalent} \\ \Gamma_{gen}, \Gamma \cup \{A :_{\Sigma} \{SP\}\} \vdash UT :: \Gamma', B \\ B : \mathbf{Sig}[SP_2] \text{ in } \Gamma' \end{array} \text{ for any family of models } \mathcal{M} \text{ compatible with } \Gamma', M_B \in \mathbf{Mod}(SP_2)}{\Gamma_{gen}, \Gamma \vdash \lambda A : SP \bullet UT :: SP_1 \rightarrow SP_2}$$

$$\Gamma_{gen}, \Gamma \vdash UT :: \Gamma', A$$

(constructive)

$$\frac{}{\Gamma_{gen}, \Gamma \vdash A :: \Gamma, A}$$

$$\frac{\begin{array}{l} A :_{\Sigma_f \rightarrow \Sigma_r} SP \rightarrow SP_r \text{ in } \Gamma_{gen} \\ \Gamma_{gen}, \Gamma \vdash UT :: \Gamma_a, A_a \\ \text{for any family of models } \mathcal{M} \text{ compatible with } \Gamma_a, M_{A_a} \in \mathbf{Mod}(SP_f) \\ (\Delta, \sigma(\Delta), \iota) \text{ is the selected pushout of } (\sigma, \iota_{\Sigma_f \subseteq \Sigma_r}) \\ A_f, A_r, B \notin \text{dom}(\Gamma_a) \end{array}}{\Gamma_{gen}, \Gamma \vdash A [UT \text{ fit } \sigma : \Sigma_f \rightarrow \Sigma_a] :: \Gamma_a \cup \{A_f :_{\Sigma_f} \{SP\}, \sigma : A_f \rightarrow A_a, \\ A_r :_{\Sigma_r} \{SP_r\}, \iota_{\Sigma_f \subseteq \Sigma_r} : A_f \rightarrow A_r, B :_{\Delta} \emptyset, \iota : A_a \rightarrow B, \sigma(\Delta) : A_r \rightarrow B\}, B}$$

$$\frac{\begin{array}{l} \Gamma_{gen}, \Gamma \vdash UT_1 :: \Gamma_1, A_1 \\ \Gamma_{gen}, \Gamma \vdash UT_2 :: \Gamma_2, A_2 \\ A_1 : \Sigma_1 \text{ in } \Gamma_1 \\ A_2 : \Sigma_2 \text{ in } \Gamma_2 \\ \text{dom}(\Gamma_1) \cap \text{dom}(\Gamma_2) = \text{dom}(\Gamma) \\ B \notin \text{dom}(\Gamma_1) \cup \text{dom}(\Gamma_2) \end{array}}{\Gamma_{gen}, \Gamma \vdash UT_1 \text{ and } UT_2 :: \Gamma_1 \cup \Gamma_2 \cup \\ \{B :_{\Sigma_1 \cup \Sigma_2} \emptyset, \iota_{\Sigma_1 \subseteq \Sigma_1 \cup \Sigma_2} : A_1 \rightarrow B, \iota_{\Sigma_2 \subseteq \Sigma_1 \cup \Sigma_2} : A_2 \rightarrow B\}, B}$$

$$\frac{\begin{array}{l} \Gamma_{gen}, \Gamma \vdash UT :: \Gamma', A \\ B \notin \text{dom}(\Gamma') \end{array}}{\Gamma_{gen}, \Gamma \vdash UT \text{ with } \sigma : \Sigma \rightarrow \Sigma' :: \Gamma' \cup \{B :_{\Sigma'} \emptyset, \sigma : A \rightarrow B\}, B}$$

$$\frac{\begin{array}{l} \Gamma_{gen}, \Gamma \vdash UT :: \Gamma', A \\ B \notin \text{dom}(\Gamma') \end{array}}{\Gamma_{gen}, \Gamma \vdash UT \text{ hide } \sigma : \Sigma \rightarrow \Sigma' :: \Gamma' \cup \{B :_{\Sigma} \emptyset, \sigma : B \rightarrow A\}, B}$$

$$\frac{\begin{array}{c} \Gamma_{gen}, \Gamma \vdash UT :: \Gamma', B \\ B : \Sigma \text{ in } \Gamma' \\ A \notin \text{dom}(\Gamma') \\ \Gamma_{gen}, \Gamma \cup \{A :_{\Sigma} \emptyset, \text{id}_{\Sigma} : A \rightarrow B\} \vdash UT' :: \Gamma'', E \\ D \notin \text{dom}(\Gamma'') \end{array}}{\Gamma_{gen}, \Gamma \vdash \mathbf{local } A = UT \mathbf{ within } UT' :: \Gamma''[D/A], E[D/A]}$$

The theorem below states that the proof calculus is successful for a given unit specification if and only if the architectural specification is correct w.r.t. the generative model semantics and the units produced with it satisfy the unit specification. It follows easily from a result in [Mosses, 2004].

Theorem 11.1.1 *Let ASP be an architectural specification such that $\vdash ASP \triangleright \square$ and no generic unit declaration of ASP is inconsistent. For each unit specification USP we have that $\vdash ASP :: USP$ if and only if $\vdash ASP \Rightarrow_g AM$ for some AM such that for all $(U, BM) \in AM$, $U \in \text{Unit}[USP]$.*

We show that the second assumption of Thm. 11.1.1 is indeed necessary

Example 11.1.2 *Let us consider the following specifications:*

```

spec CONSTS =
  sort s
  ops a, b : s
spec EQCONSTS = CONSTS
  then
  • a = b
spec DIFFCONSTS = CONSTS
  then
  •  $\neg (a = b)$ 
arch spec ASP =
  units
    M : EQCONSTS;
    F : DIFFCONSTS  $\rightarrow$  EQCONSTS;
  result F[M]

```

The unit specification $\text{DIFFCONSTS} \rightarrow \text{EQCONSTS}$ is inconsistent, because the extension is obviously non-conservative, as we cannot construct a model in which the interpretation of the constants a and b are equal from a model where they are different in a way that the argument model is preserved. Therefore, ASP denotes the empty class of models. Hence, the right hand-side of the equivalence in Theorem 11.1.1 holds for any unit specification USP over the signature of CONSTS. However, the verification condition $\text{DIFFCONSTS} \rightsquigarrow \text{EQCONSTS}$ for the unit application of F does not hold, and therefore the left hand-side of the equivalence is false. This shows that the condition that generic units must be consistent is needed in Theorem 11.1.1.

We modify the proof calculus such that it becomes fully constructive: USP is no longer provided, but rather obtained by defining the specification $\mathcal{S}_{ASP}(UE)$ of each unit expression UE occurring in ASP inductively on its structure. Clearly, the specification of a unit term given by a declared unit name can be read off the declaration. If a unit term is translated, then its specification is so as well, that is, $\mathcal{S}_{ASP}(UT \text{ with } \sigma) = \mathcal{S}_{ASP}(UT) \text{ with } \sigma$. Similarly for hiding. A complication arises when determining the unit specification of a unit amalgamation or unit application. The naïve solution to define $\mathcal{S}_{ASP}(UT_1 \text{ and } UT_2)$ to be $\mathcal{S}_{ASP}(UT_1) \text{ and } \mathcal{S}_{ASP}(UT_2)$ does not work. We illustrate this with an example.

Example 11.1.3 *Let us consider the following architectural specification:*

```

arch spec ASP =
  units
    UN : sort s;
    UT = (UN with s ↦ t) and (UN with s ↦ u)
  result UT
end

```

Then $\mathcal{S}_{ASP}(UN) = (\text{sort } s; \emptyset)$. The naïve definition of $\mathcal{S}_{ASP}(UT)$ would be $(\text{sorts } t, u; \emptyset)$. Now a model of ASP is a pair (U, BM) where $BM(UN)$ is a model of $(\text{sort } s; \emptyset)$ and $BM(UT)$ interprets both the sort t and the sort u as $BM(UN)_s$. This requirement is not recorded in $(\text{sorts } t, u; \emptyset)$; hence, this specification is too weak. In order to express that sorts t and u must be interpreted in the same way, we need to look at the following diagram:

$$\begin{array}{ccc}
 s & \xrightarrow{s \mapsto t} & t \\
 s \mapsto u \downarrow & & \downarrow \\
 u & \longrightarrow & t, u \xrightarrow[t, u \mapsto s]{\eta_{UT}} s
 \end{array}$$

Now $(\text{sorts } t, u; \emptyset) \text{ and } (\text{sort } s; \emptyset) \text{ hide } t, u \mapsto s$ is the desired specification; it requires that sorts t and u are interpreted in the same way.

In general, we will need to keep track of the dependencies between symbols, using a weakly amalgamable cocone of the diagram of the unit term. The existence of the latter is ensured by a successful run of the extended static semantics. This is captured by the following definition:

Definition 11.1.4 *For each unit term UT , let us denote D_{UT} the sub-diagram corresponding to the node of UT in the context Γ of an architectural specification¹. Then we define $\mathcal{S}_{colim}(UT) = (\Sigma, \emptyset) \text{ hide } \eta_A$ where A is the node of the unit term UT in D_{UT} and $\eta = (\Sigma, \{\eta_X\}_{X \in \text{dom}(D_{UT})})$ is a weakly amalgamable cocone for D_{UT} .*

¹This can be understood as restricting the declarations and definitions of the units of the architectural specification to those involved in UT , and then constructing the context Γ with the rules of the proof calculus.

We make use of an auxiliary structure for storing the specifications of the units declared and defined in an architectural specification.

Definition 11.1.5 A verification context is a finite map Γ_v assigning unit specifications to unit names.

We denote the empty verification context by Γ_v^\emptyset .

The specification of a unit expression is then defined relatively to a verification context.

Definition 11.1.6 Let Γ_v be a verification context and UE a unit expression. Then the specification of UE w.r.t Γ_v , denoted $\mathcal{S}_{\Gamma_v}(UE)$, is defined as follows:

- if UE is a unit term UT , $\mathcal{S}_{\Gamma_v}(UT)$ is defined inductively:
 - if UT is a unit name, then $\mathcal{S}_{\Gamma_v}(UT) = SP$ where $\Gamma_v(UT) = SP$;
 - if $\mathcal{S}_{\Gamma_v}(UT) = SP$, then $\mathcal{S}_{\Gamma_v}(UT \text{ with } \sigma) = SP \text{ with } \sigma$;
 - if $\mathcal{S}_{\Gamma_v}(UT) = SP$, then $\mathcal{S}_{\Gamma_v}(UT \text{ hide } \sigma) = SP \text{ hide } \sigma$;
 - if $UT = UT_1 \text{ and } UT_2$ and $\mathcal{S}_{\Gamma_v}(UT_i) = SP_i$ for $i = 1, 2$ then $\mathcal{S}_{\Gamma_v}(UT) = SP_1 \text{ and } SP_2 \text{ and } \mathcal{S}_{colim}(UT)$;
 - if $UT = F[UT_1 \text{ fit } \sigma_1] \dots [UT_n \text{ fit } \sigma_n]$, where $\Gamma_v(F) = SP_1 \times \dots \times SP_n \rightarrow SP$ and for any $i = 1, \dots, n$, $SP_i \text{ with } \sigma_i \rightsquigarrow \mathcal{S}_{\Gamma_v}(UT_i)$, then $\mathcal{S}_{\Gamma_v}(UT) = \{SP \text{ with } \sigma\} \text{ and } \mathcal{S}_{\Gamma_v}(UT_1) \text{ with } \iota_1; \iota' \text{ and } \dots \text{ and } \mathcal{S}_{\Gamma_v}(UT_n) \text{ with } \iota_n; \iota' \text{ and } \mathcal{S}_{colim}(UT)$, where the application is done as in the diagram below, with $\Sigma_f = \cup_{i=1, \dots, n} \mathbf{Sig}[SP_i]$, $\Sigma_r = \mathbf{Sig}[SP]$, $\Sigma_i = \mathbf{Sig}[UT_i]$, $\Sigma_a = \cup_{i=1, \dots, n} \Sigma_i$, all ι are inclusions, $\sigma = \cup_{i=1, \dots, n} \sigma_i : \Sigma_f \rightarrow \Sigma_a$, and $(\Sigma, \iota', \sigma')$ is the selected pushout of (ι, σ) (see Dfn.4.4.1) :

$$\begin{array}{ccc}
 \Sigma_f & \xrightarrow{\iota} & \Sigma_r \\
 \sigma \downarrow & & \sigma' \downarrow \\
 \Sigma_a & \xrightarrow{\iota'} & \Sigma \\
 \iota_1 \nearrow & & \nwarrow \iota_n \\
 \Sigma_1 & \dots & \Sigma_n
 \end{array}$$

- $\mathcal{S}_{\Gamma_v}(\text{local UDEFN within } UT) = \mathcal{S}_{\Gamma'_v}(UT)$, where Γ'_v extends Γ_v according to UDEFN in the obvious way.
- if UE is a lambda expression $\lambda X : SP . UT$, then $\mathcal{S}_{\Gamma_v}(UE) = SP \rightarrow \mathcal{S}_{\Gamma'_v}(UT)$, where Γ'_v extends Γ_v with $X : SP$.

If Γ_v has been built from all the units declared and defined in an architectural specification ASP , we may denote the specification of a unit expression UE by

$\mathcal{S}_{ASP}(UE)$ instead of $\mathcal{S}_{\Gamma_v}(UE)$. Moreover, we denote by \mathcal{S}_{ASP} the specification of the result unit expression in *ASP*.

Notice that if $\vdash UT \triangleright \Sigma_{UT}$ and no non-generic unit is used more than once in *UT*, directly or indirectly via unit definitions, we have that $\mathcal{S}_{colim}(UT) = \Sigma_{UT} \mathbf{hide} id_{\Sigma_{UT}}$. This allows us to omit $\mathcal{S}_{colim}(UT)$ from $\mathcal{S}_{ASP}(UT)$ in such cases.

In general, the specification of a unit term does not give an exact axiomatization of the class of models of the unit term. The reason is that non-generativity of CASL architectural semantics can not always be captured by a structured specification, and this becomes visible when the unit term involves more than one application of the same generic unit, as we can see from Ex. 11.1.7 below. However, we will use $\mathcal{S}_{\Gamma_v}(UT)$ as an approximation, since models of the unit term are also models of this specification.

Example 11.1.7 [Hoffman, 2005] Consider the following architectural specification:

```

arch spec ASP =
  units
    A : {sort s};
    F : {sort s} → {sort s; op a : s};
    G : {sort s} → {sort s; op a : s};
    H : {sort s; op a : s} → {sort s; ops a, b : s};
    B = F [A];
    C = G [A]
  result H [B] and {H [C] with a ↦ a', b ↦ b'}
end

```

On one side, the specification \mathcal{S}_{ASP} of the result unit term of *ASP* is $\{\mathbf{sort} \ s; \mathbf{ops} \ a, b, a', b' : s\}$. On the other side, if M is a model of \mathcal{S}_{ASP} such that $M_a = M_{a'}$, the non-generative semantics imposes that applying *H* twice (with *B* and *C* as arguments, respectively) to the same model yields the same result, and therefore M_b and $M_{b'}$ must also be equal. However, $a = a' \implies b = b'$ is not a consequence of \mathcal{S}_{ASP} , which is in this case only an over-approximation of the model class of the result unit term of *ASP*.

We are now ready to introduce a new calculus for correctness of architectural specifications, that we refer to as *constructive*, as in Fig. 11.2. The judgements of the calculus are of the form $\vdash ASP ::_c USP$, where *USP* is now no longer provided as an argument, but constructed by the rules of the calculus. Therefore, the only verification conditions of this calculus are those introduced in the definition of the specification of a unit expression. It is also no longer needed to carry dependencies between units in a diagram labeled with specifications. Instead, the calculus builds a verification context for the units declared or defined, and this verification context is then used to construct the specification of the result unit expression of the architectural specification being verified. The rule for unit declarations takes

$$\begin{array}{c}
\frac{\Gamma_v^\emptyset \vdash ASP ::_c USP}{\vdash \mathbf{arch\ spec}\ ASP ::_c USP} \\
\\
\Gamma \vdash UDD_1 ::_c \Gamma_1 \\
\vdots \\
\Gamma_{n-1} \vdash UDD_n ::_c \Gamma_n \\
\hline
\Gamma \vdash \mathbf{units}\ UDD_1 \dots UDD_n \mathbf{result}\ UE ::_c \mathcal{S}_{\Gamma_n}(UE) \\
\\
\frac{\Gamma \vdash UDECL ::_c \Gamma'}{\Gamma \vdash UDECL \mathbf{qua}\ UDD ::_c \Gamma'} \qquad \frac{\Gamma \vdash UDEFN ::_c \Gamma'}{\Gamma \vdash UDEFN \mathbf{qua}\ UDD ::_c \Gamma'} \\
\\
\frac{\Gamma \vdash USP ::_c USP'}{\Gamma \vdash UN : USP ::_c \Gamma \cup \{UN \mapsto USP'\}} \\
\\
\frac{}{\Gamma \vdash UN = UE ::_c \Gamma \cup \{UN \mapsto \mathcal{S}_\Gamma(UE)\}} \\
\\
\hline
\Gamma \vdash SP_1 \times \dots \times SP_n \rightarrow SP ::_c SP_1 \times \dots \times SP_n \rightarrow SP
\end{array}$$

Figure 11.2: Proof calculus for CASL architectural specifications.

into account the fact that the specification USP of a unit can be itself architectural. The result of applying the calculus to USP is a unit type USP' , which can be either USP if USP was already a unit type (last rule of the calculus) or the specification of the result unit of USP if USP is architectural. We moreover use the context of outer declarations of definitions when verifying the architectural specification USP' . This is done to provide support for unit imports, see Sec. 10.5.

In the sequel we will use the following framework.

Framework 11.1.8 *ASP is an architectural specification such that $\vdash ASP \triangleright \square$ and no generic unit is inconsistent.*

The constructive and the deductive versions of the proof calculus are related by the following result.

Theorem 11.1.9 *In conditions of Framework 11.1.8, $\vdash ASP ::_c USP$ implies $\vdash ASP :: USP$.*

Proof. Assume that $\mathbf{arch\ spec}\ ASP = \mathbf{units}\ UDD^+ \mathbf{result}\ UE$. Moreover, assume $\Gamma_v^\emptyset \vdash UDD^+ ::_c \Gamma_v$ and $\vdash UDD^+ :: \Gamma_{gen}, \Gamma$. We prove that if $USP = \mathcal{S}_{ASP}$ then $\Gamma_{gen}, \Gamma \vdash UE :: USP$.

We make a case distinction on UE . If $UE = UT$, it suffices to show that for any family of models M compatible with the diagram Γ' , $M_B \models \mathcal{S}_{ASP}(UT)$, where

$\Gamma_{gen}, \Gamma \vdash UT :: \Gamma', B$. We will proceed by induction on the structure of UT . Notice that in the case of unit application and union, $M_B \models \mathcal{S}_{colim}(B)$ using the amalgamation properties ensured by the successful run of the extended static semantics.

- If $UT = UN$, then by definition $\mathcal{S}_{ASP}(UT) = SP$ where $UN : SP$ in ASP and by construction of Γ' , the node UN is labeled with SP . This means that $M_{UN} \models SP = \mathcal{S}_{ASP}(UT)$.
- If $UT = UT_1$ and UT_2 , then by construction B has two incoming edges from A_1 and A_2 , where $\Gamma_{gen}, \Gamma \vdash UT_i :: \Gamma_i, A_i, i = 1, 2$. By the inductive hypothesis $M_{A_i} \models \mathcal{S}_{ASP}(UT_i)$. It follows easily that $M_B \models \mathcal{S}_{ASP}(UT_1)$ and $\mathcal{S}_{ASP}(UT_2)$.
- If $UT = F[UT' \text{ fit } \sigma]$, then, with the notation of the rule, B has two incoming edges from A_r and A_a by construction, A_r is labeled with SP_r , and $M_{A_a} \models \mathcal{S}_{ASP}(UT')$ by induction. It is easy to check that $M_B \models \mathcal{S}_{ASP}(UT)$.
- If $UT = UT'$ with σ then, with the notation of the rule, B has an incoming edge from A by construction and $M_A \models \mathcal{S}_{ASP}(UT')$, which implies $M_B \models \mathcal{S}_{ASP}(UT)$.
- If $UT = UT'$ hide σ then, with the notation of the rule, B has an outgoing edge to A by construction and $M_A \models \mathcal{S}_{ASP}(UT')$, which implies $M_B \models \mathcal{S}_{ASP}(UT)$.

If $UE = \lambda X : SP . UT$, we have $M_B \models \mathcal{S}_{ASP}(UT)$ where $\Gamma_{gen}, \Gamma \vdash UT :: \Gamma', B$ and since SP is obviously equivalent with itself, we have that $\Gamma_{gen}, \Gamma \vdash UE :: SP \rightarrow \mathcal{S}_{ASP}(UT)$.

□

Together with Thm. 11.1.1, we get the following immediate result.

Theorem 11.1.10 (Soundness) *Under requirements of Framework 11.1.8, if $\vdash ASP ::_c USP$, we have that $\vdash ASP \Rightarrow_g \mathcal{AM}$ and for all $(U, BM) \in \mathcal{AM}$, $U \in \text{Unit}(USP)$.*

For the implication in the other direction, we need to strengthen the framework, because the specification of the unit term just approximates its model class:

Framework 11.1.11 *ASP is an architectural specification such that $\vdash ASP \triangleright \square$, each generic unit is consistent and is not applied more than once, and each non-generic unit not being used (directly or indirectly) in the result unit expression is consistent as well.*

Let us first notice that in some cases the obtained unit specification exactly captures the models of the architectural specification, if the latter are projected with the possible semantics for the result unit term. Let therefore *ProjRes* take any model of an architectural specification to the interpretation of its result unit term in this model.

Theorem 11.1.12 *Under requirements of Framework 11.1.11, if $\vdash ASP ::_c USP$ then $ProjRes(\mathbf{Mod}(ASP)) = \mathbf{Mod}(USP)$.*

Proof. The inclusion from left to right follows from Thm. 11.1.1 and Thm. 11.1.10. Concerning the converse inclusion, let $\vdash ASP \triangleright \square$ and $\vdash ASP ::_c USP$. We distinguish the cases of ASP being consistent or not. If ASP is inconsistent, the specification of some non-generic unit UN in ASP must be inconsistent, because all generic unit specification have been assumed to be consistent. But then UN is being used in the result unit expression of ASP , causing USP to be inconsistent as well. Hence, both sides of the equation are the empty set.

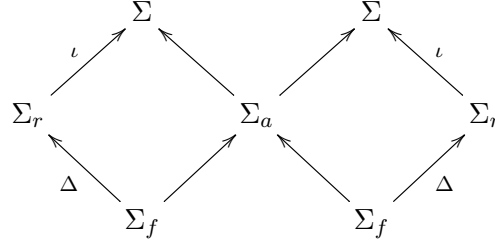
Let us now assume that ASP is consistent, and let M be a model of USP . For simplicity, we consider M to be non-parametric; the parametric case is treated similarly by considering all the possible applications to argument units. Let UT be the result unit term of ASP . For each occurrence of a subterm oT of UT in UT , we construct a model M_{oT} satisfying $\mathcal{S}_{ASP}(T)$, where the latter is built in the context of the unit definitions of ASP . In parallel, we define partial functions M_F for parametric units F for one or more argument tuples. Let h compute the height of a unit term. We proceed by nested induction over $h(UT) - h(oT)$ within unit terms and the dependency structure of the unit definitions, which means by moving from each subterm to its immediate subterms and in the order of occurrence of defined units in the unit term.

The induction base is UT ; we put $M_{UT} := M$. For the induction step, we make a case distinction:

- if T is a unit name coming from a unit declaration, then there are no subterms;
- if T is a unit name coming from a unit definition $T = UT'$, we repeat the procedure for UT' with the model $M_T = M_{UT'}$;
- if T is T_1 **and** T_2 , let $M_{oT_i} := M_T|_{\sigma_i}$, where σ_i is the inclusion of the signature of T_i into that of T . From $M_T \in \mathbf{Mod}(\mathcal{S}_{ASP}(T))$, we easily get $M_{oT_i} \in \mathbf{Mod}(\mathcal{S}_{ASP}(T_i))$.
- if T is T_1 **with** σ , let $M_{oT_1} := M_T|_{\sigma}$. Again, from $M_T \in \mathbf{Mod}(\mathcal{S}_{ASP}(T))$, we easily get $M_{oT_1} \in \mathbf{Mod}(\mathcal{S}_{ASP}(T_1))$;
- if T is T_1 **hide** σ , $\mathcal{S}_{ASP}(T)$ is $\mathcal{S}_{ASP}(T_1)$ **hide** σ . Hence, by $M_T \in \mathbf{Mod}(\mathcal{S}_{ASP}(T))$, there is some (not necessarily unique) $M_1 \in \mathbf{Mod}(\mathcal{S}_{ASP}(T_1))$ with $M_1|_{\sigma} = M_T$. Put $M_{oT_1} := M_1$;
- if $T = F[T_1 \text{ fit } \sigma_1] \dots [T_n \text{ fit } \sigma_n]$, then we know that $\mathcal{S}_{ASP}(T) = \{SP \text{ with } \sigma\}$ **and** $\mathcal{S}_{ASP}(UT_1)$ **with** $\iota_1; \iota'$ **and** \dots **and** $\mathcal{S}_{ASP}(UT_n)$ **with** $\iota_n; \iota'$ **and** $\mathcal{S}_{colim}(UT)$. Hence, for $i = 1, \dots, n$, we define $M_{oT_i} := M|_{\iota_i; \iota'}$. We also define the action of M_F on the arguments thus defined: $M_F(M_{oT_1}|_{\sigma_1}, \dots, M_{oT_n}|_{\sigma_n}) := M_T$. (Note that by assumption, F is applied only once.) If F is the name of a generic unit coming from a unit definition, we repeat the procedure for the term defining F .

We need to show that if oT_1 and oT_2 are two occurrences of a term T in UT , then $M_{oT_1} = M_{oT_2}$. Thus, we can define $M_T = M_{oT}$ for each subterm T of UT and for an arbitrary occurrence oT of T in UT and we also ensure that M_F is well-defined, for any generic unit F .

Assume that oT_1 and oT_2 are two occurrences of T in UT . Let UT' be the least subterm of UT that contains both oT_1 and oT_2 , and notice that UT' can be either a unit application $F[UT_1] \dots [UT_n]$ with oT_1, oT_2 subterms of UT_i and UT_j respectively, for some $i, j \in 1, \dots, n$ or a unit amalgamation UT_1 **and** \dots **and** UT_n with oT_1, oT_2 subterms of UT_i and UT_j respectively, for some $i, j \in 1, \dots, n$. In both cases, by Def. 11.1.6 we have that $M_{UT'} \models \mathcal{S}_{colim}(UT')$. Since the framework assumes that any generic unit can be applied only once, T cannot be a unit application. If that were the case, the diagram of the unit term $D_{UT'}$ would contain a sub-diagram like the following:



where the nodes labeled with Σ correspond to the two occurrences oT_1 and oT_2 . Then the symbols of Σ_r that are not in the image along Δ of the symbols in Σ_f do not have a common origin in the diagram, and therefore their image along σ gives symbols that do not necessarily share.

In all other possible cases, we can observe that the nodes corresponding to oT_1 and oT_2 in the diagram $D_{UT'}$ of the unit term UT' have symbols with the same origin. This implies that M_{oT_1} and M_{oT_2} must be equal.

We now construct a model of ASP as follows: non-parametric unit names A are interpreted as M_A (if this is defined), while parametric units F are interpreted as M_F whenever this is defined for specific arguments. The interpretations of the remaining non-parametric units and remaining applications of parametric units to arguments does not affect UT at all, we hence can take them from any model of ASP (which exists by consistency of ASP). Altogether, we have constructed a model of ASP that interprets UT as M .

□

Theorem 11.1.13 *Under requirements of Framework 11.1.11, if $\vdash ASP :: USP$ for some USP , then $\vdash ASP ::_c USP'$ where $USP' = \mathcal{S}_{ASP}$ is the specification of the result unit expression UE of ASP and moreover $USP \rightsquigarrow USP'$.*

Proof.

Assume that **arch spec** $ASP = \mathbf{units} \ UDD^+ \ \mathbf{result} \ UE$. Moreover, assume $\vdash UDD^+ :: \Gamma_{gen}, \Gamma$. We prove that if for any unit expression UE' occurring in ASP we

have $\Gamma_{gen}, \Gamma \vdash UE' :: USP$ for some USP , then $\Gamma_v^\emptyset \vdash UDD^+ ::_c \Gamma_v, \Gamma_v \vdash UE' ::_c USP'$ where $USP' = \mathcal{S}_{\Gamma_v}(UE')$ and $USP \rightsquigarrow USP'$. Then the theorem follows immediately by taking $UE' = UE$.

The proof is done by induction on pairs of the form (x, UE') where UE' is a unit expression occurring in ASP and x is the position (given as a natural number) of the occurrence of UE' in ASP , ordered lexicographically: $(x_1, UE_1) < (x_2, UE_2)$ iff $x_1 < x_2$, or $x_1 = x_2$ and UE_1 is a subexpression of UE_2 .

For the induction base we have to consider pairs $(1, UN)$ where $UN : USP'$ in ASP and UN is a unit name that appears in the unit expression UE involved in the first unit definition of ASP . By the hypothesis we have that $\Gamma_{gen}, \Gamma \vdash UN :: USP$ for some USP . By definition of $\mathcal{S}_{ASP}(UN)$, we have that $\Gamma_v \vdash UN ::_c USP'$. We have to show that $USP \rightsquigarrow USP'$. Let ASP' be the architectural specification obtained from ASP by replacing its result unit expression with UN . Since $\vdash ASP' ::_c USP'$, with Thm. 11.1.12 we have $ProjRes(\mathbf{Mod}(ASP')) = \mathbf{Mod}(USP')$. By $\vdash ASP' :: USP$ and soundness of $\vdash _ :: _$ we have $ProjRes(\mathbf{Mod}(ASP')) \subseteq \mathbf{Mod}(USP)$. Hence, $\mathbf{Mod}(USP') \subseteq \mathbf{Mod}(USP)$.

For the inductive step, we assume the property holds for each $(x_0, UE_0) < (x, UE)$ and we want to show it for (x, UE) , where x is the position of the occurrence of $UN = UE$ in ASP . We make a case analysis on UE , and we present here just the case of unit applications; the others are similar. Therefore, let us assume that $UE = F[UE']$ where $F : SP \rightarrow SP'$ in Γ_{gen} . Since $\Gamma_{gen}, \Gamma \vdash UE :: USP$ for some USP , we can easily show that $\Gamma_{gen}, \Gamma \vdash UE' :: SP$ (this follows directly from the verification condition of the proof calculus $\vdash _ :: _$ for unit applications). By the inductive hypothesis, we get that $\Gamma_v \vdash UE' ::_c \mathcal{S}_{ASP}(UE')$ and moreover $SP \rightsquigarrow \mathcal{S}_{ASP}(UE')$, which is precisely the verification condition of the proof calculus $\vdash _ ::_c _$ for $F[UE']$. This means that $\Gamma_v \vdash UE ::_c \mathcal{S}_{ASP}(UE)$, where $\mathcal{S}_{ASP}(UE)$ is as given in Def. 11.1.6. We need to show that $USP \rightsquigarrow \mathcal{S}_{ASP}(UE)$, and this is done with the same argument as for the inductive base. \square

We now can combine the results that we have obtained so far. Therefore, Thm. 11.1.1 needs to be generalized to the whole architectural language. Recall that unit imports are excluded. This means we have to treat unit definitions and generalise from single parameter to multiparameter units. This is rather straightforward. We then get the following result.

Theorem 11.1.14 (Completeness) *Under requirements of Framework 11.1.11, if $\vdash ASP \Rightarrow_g AM$ for some AM such that for all $(U, BM) \in AM$, $U \in Unit[USP]$ for some USP then $\vdash ASP ::_c USP'$, where $USP' = \mathcal{S}_{ASP}$ is the specification of the result unit expression UE of ASP and moreover $USP \rightsquigarrow USP'$.*

11.2 Proof Calculus for CASL Refinements

While the proof calculus for architectural specifications checks that result units satisfy a given unit specification, we introduce a counterpart for the specification that

refinements should satisfy, tailored according to the three kinds of refinements. This allows to express the proof calculus rule for compositions of refinements in a more concise manner.

Definition 11.2.1 *Let $R\Sigma$ be a refinement signature. A refinement specification S over $R\Sigma$ is defined as follows:*

- if $R\Sigma = (U\Sigma_1, U\Sigma_2)$, then S takes the form (USP_1, USP_2) such that $\vdash USP_i \triangleright U\Sigma_i$, for $i=1,2$;
- if $R\Sigma = (U\Sigma, B\Sigma)$, then S takes the form (USP, BSP) , where $\vdash USP \triangleright U\Sigma$ and BSP is a branching specification, which is in turn either a unit specification USP' such that $\vdash USP' \triangleright U\Sigma'$, when $B\Sigma = U\Sigma'$ or a map \mathcal{SPM} such that $\text{dom}(\mathcal{SPM}) = \text{dom}(BstC)$ and $\mathcal{SPM}(X)$ is a branching specification over $BstC(X)$, for each $X \in \text{dom}(BstC)$, when $B\Sigma = BstC$;
- if $R\Sigma = \{UN_i \mapsto R\Sigma_i\}_{i \in \mathcal{J}}$, then S takes the form $\{UN_i \mapsto S_i\}_{i \in \mathcal{J}}$, where S_i is a refinement specification over $R\Sigma_i$.

The intuitive idea is that if a refinement specifies a constructor $\kappa : \mathbf{Mod}(\Sigma) \rightarrow \mathbf{Mod}(\Sigma')$, a refinement specification (USP', USP) consists of two unit specifications USP and USP' with $\vdash USP \triangleright \Sigma$ and $\vdash USP' \triangleright \Sigma'$ such that $\text{dom}(\kappa) \subseteq \mathbf{Mod}(USP)$ and $\text{cod}(\kappa) \subseteq \mathbf{Mod}(USP')$. The latter two conditions are captured by Def. 11.2.3. This generalizes to n -ary constructors and to families of constructors in the obvious way. We denote the empty map with \mathcal{SPM}_\emptyset .

Again, the proof calculus rules rely on a composition operation for refinement specifications.

Definition 11.2.2 *Let $R\Sigma_1$ and $R\Sigma_2$ be two refinement signatures such that $R\Sigma_1; R\Sigma_2$ is defined and let S_i be a refinement specification over $R\Sigma_i$ for $i = 1, 2$. The composition $S_1; S_2$ is defined inductively as follows:*

- if $S_1 = (USP_1, USP_2)$, then $S_2 = (USP_3, BSP)$ and $S_1; S_2 = (USP_1, BSP)$, provided that $USP_2 \rightsquigarrow USP_3$.
- if $S_1 = (USP_1, \mathcal{SPM}_1)$ then S_2 must be of the form $\{UN_i \mapsto S_i\}_{i \in \mathcal{J}}$. We define $S_1; S_2 = (USP_1, \mathcal{SPM}_1[S_2])$, where $\mathcal{SPM}[S]$ modifies \mathcal{SPM} for any $A \in \text{dom}(S)$ as follows:
 - if $\mathcal{SPM}(A)$ is a unit specification USP , then $S(A)$ must be of the form (USP', BSP') . Then $\mathcal{SPM}[S](A) = BSP'$, provided that $USP \rightsquigarrow USP'$.
 - if $\mathcal{SPM}(A) = \mathcal{SPM}'$, then $S(A)$ must be of the form $\{UN'_j \mapsto S'_j\}_{j \in \mathcal{J}'}$. Then $\mathcal{SPM}[S](A) = \mathcal{SPM}'[S(A)]$.
- if $S_1 = \{UN_i \mapsto S_i\}_{i \in \mathcal{J}}$, then $S_1; S_2$ is defined only if $S_2 = \{UN'_j \mapsto S'_j\}_{j \in \mathcal{J}'}$. Then $S_1; S_2$ modifies the ill-defined union of S_1 and S_2 by putting $(S_1; S_2)(A) = S_1(A); S_2(A)$ for any $A \in \text{dom}(S_1) \cap \text{dom}(S_2)$.

The constructive proof calculus for architectural specifications is complemented at the level of refinements as in Fig. 11.3. The judgments of the proof calculus for refinements are of the form $\vdash SPR ::_c S$, where SPR is a refinement and S is a refinement specification and the calculus is defined inductively on the structure of the refinements. We also complete the architectural proof calculus, as introduced in Fig. 11.2, as specifications of units are now simple or branching refinements. The idea is that for each unit declaration $UN_i : SPR_i$ of an architectural specification ASP , we define $\Gamma_v(UN_i) = USP_i$ if $\vdash SPR_i ::_c (USP_i, BSP_i)$, where Γ_v is the verification context of ASP . This would suffice for the verification of ASP with the architectural calculus. Furthermore, we need to set $SPM(UN_i) = BSP_i$ in the refinement specification SPM of ASP , to be able to check correctness of further refinements of the units UN_i .

Definition 11.2.3 Let $R\Sigma$ be a refinement signature, S a refinement specification over $R\Sigma$ and \mathcal{R} a refinement relation over $R\Sigma$. We define the satisfaction of a refinement specification by a refinement relation, denoted $\mathcal{R} \models S$, inductively as follows:

- if $R\Sigma = (U\Sigma, U\Sigma')$, then $S = (USP, USP')$ and $\mathcal{R} \subseteq \{(U, U') \mid U \in \text{Unit}(U\Sigma), U' \in \text{Unit}(U\Sigma')\}$. Then $\mathcal{R} \models S$ iff $U \in \text{Unit}(USP)$ and $U' \in \text{Unit}(USP')$ for any $(U, U') \in \mathcal{R}$ and moreover, for any $U' \in \text{Unit}(USP')$ there is a $U \in \text{Unit}(USP)$ such that $(U, U') \in \mathcal{R}$;
- if $R\Sigma = (U\Sigma, BstC)$, then $S = (USP, SPM)$ and $\mathcal{R} \subseteq \{(U, BM) \mid U \in \text{Unit}(U\Sigma), BM \text{ is a branching model over } BstC\}$. Then $\mathcal{R} \models S$ iff for any $(U, BM) \in \mathcal{R}$, $U \in \text{Unit}(USP)$ and for any $A \in \text{dom}(SPM)$ we have that $BM(A) \models SPM(A)$. (SPM and BM have the same domain) and moreover for each branching model BM of SPM there is a unit U such that $(U, BM) \in \mathcal{R}$;
- if $R\Sigma = \{UN_i \mapsto R\Sigma_i\}_{i \in \mathcal{J}}$, then $S = \{UN_i \rightarrow S_i\}_{i \in \mathcal{J}}$ and $\mathcal{R} = \{UN_i \rightarrow \mathcal{R}_i\}_{i \in \mathcal{J}}$. Then $\mathcal{R} \models S$ iff $\mathcal{R}_i \models S_i$ for any $i \in \mathcal{J}$.

The following result states that if a statically well-formed refinement SPR can be proven correct w.r.t. a refinement specification S using the proof calculus for refinements, then SPR has a denotation according to the model semantics and moreover the refinement relation thus obtained satisfies S .

Theorem 11.2.4 (Soundness) Let SPR be a refinement such that $\vdash SPR \triangleright \square$ and all refinements appearing in SPR are consistent. If $\vdash SPR ::_c S$, then $\vdash SPR \Rightarrow \mathcal{R}$ for some refinement relation \mathcal{R} and moreover $\mathcal{R} \models S$.

Proof.

The proof follows by induction on the structure of SPR .

If $SPR = USP$, then according to the model semantics rules we have $\vdash SPR \Rightarrow \mathcal{R}$ where $\mathcal{R} = \{(U, U) \mid U \in \text{Unit}(USP)\}$. With the proof calculus rule we have $\vdash SPR ::_c (USP, USP)$ and it is obvious that $\mathcal{R} \models (USP, USP)$.

$$\begin{array}{c}
\frac{}{\vdash USP ::_c (USP, USP)} \quad \frac{\vdash SPR ::_c (USP', BSP) \quad USP \rightsquigarrow_{\sigma} USP'}{\vdash USP \text{ refined via } \sigma \text{ to } SPR ::_c (USP, BSP)} \\
\\
\frac{\vdash SPR_1 ::_c S_1 \quad \vdash SPR_2 ::_c S_2 \quad S = S_1; S_2}{\vdash SPR_1 \text{ then } SPR_2 ::_c S} \quad \frac{\text{for each } i \in \mathcal{J} \vdash SPR_i ::_c S_i}{\vdash \{UN_i \text{ to } SPR_i\}_{i \in \mathcal{J}} ::_c \{UN_i \rightarrow S_i\}_{i \in \mathcal{J}}} \\
\\
\frac{\frac{\Gamma_{\emptyset}, SPM_{\emptyset} \vdash ASP ::_c (USP, SPM)}{\vdash \text{arch spec } ASP ::_c (USP, SPM)}}{\Gamma, SPM \vdash UDD_1 ::_c (\Gamma_1, SPM_1)} \\
\vdots \\
\frac{\Gamma_{n-1}, SPM_{n-1} \vdash UDD_n ::_c (\Gamma_n, SPM)}{\Gamma \vdash \text{units } UDD_1 \dots UDD_n \text{ result } UE ::_c (S_{\Gamma_n}(UE), SPM)} \\
\\
\frac{\Gamma, SPM \vdash UDECL ::_c (\Gamma', SPM')}{\Gamma, SPM \vdash UDECL \text{ qua } UDD ::_c (\Gamma', SPM')} \\
\\
\frac{\Gamma \vdash UDEFN ::_c \Gamma'}{\Gamma, SPM \vdash UDEFN \text{ qua } UDD ::_c (\Gamma', SPM)} \\
\\
\frac{\Gamma \vdash SPR ::_c (USP, BSP)}{\Gamma, SPM \vdash UN : SPR ::_c (\Gamma \cup \{UN \mapsto USP\}, SPM \cup \{UN \mapsto BSP\})} \\
\\
\frac{}{\Gamma \vdash UN = UE ::_c \Gamma \cup \{UN \mapsto S_{\Gamma}(UE)\}}
\end{array}$$

Figure 11.3: Proof calculus for CASL refinements.

If $SPR = USP$ refined via σ to SPR' such that $\vdash SPR ::_c S$, then according to the proof calculus rule we have that $\vdash SPR' ::_c (USP', BSP)$, $USP \rightsquigarrow^\sigma USP'$ and $S = (USP, BSP)$. By the inductive hypothesis we get that there is \mathcal{R}' such that $\vdash SPR \Rightarrow \mathcal{R}'$ and $\mathcal{R}' \models (USP', BSP)$. Then $\vdash SPR \Rightarrow \mathcal{R}$ where $\mathcal{R} = \{(U|_\sigma, BM) \mid (U, BM) \in \mathcal{R}'\}$ and the refinement condition $USP \rightsquigarrow^\sigma USP'$ ensures that $\mathcal{R} \models S$.

If $SPR = \mathbf{arch\ spec}\ ASP$, then according to the proof calculus rule we have that for any unit declaration of the form $UN_i : SPR_i$ in ASP , $\vdash SPR_i ::_c (USP_i, BSP_i)$. Let ASP' be the architectural specification obtained by replacing SPR_i with USP_i for each unit declaration $UN_i : SPR_i$ in ASP and let UE denote the result unit of ASP' . Then we define $SPM(UN)$ for any unit UN of ASP as $SPM(UN) = BSP_i$, if UN is a declared unit UN_i of ASP , and $SPM(UN) = S_{\Gamma_v}(UE')$ if $UN = UE'$ is a unit definition of ASP , where Γ_v is the verification environment of ASP' . This allows us to define $S = (S_{\Gamma_v}(UE), SPM)$. By induction we have $\vdash SPR_i \Rightarrow \mathcal{R}_i$ and $\mathcal{R}_i \models (USP_i, BSP_i)$ for each i . Then $\vdash SPR \Rightarrow \mathcal{R}$ where $\mathcal{R} = \{(U, \pi_2(RE)) \mid RE(UN_i) \in \mathcal{R}_i \text{ for the defined units } UN_i, U \text{ combines the units in } \pi_1(RE) \text{ according to } UE\}$ and $\mathcal{R} \models S$ by Thm. 11.1.10, where $\mathcal{AM} = \mathcal{R}$.

If $SPR = \{UN_i \mathbf{to}\ SPR_i\}_{i \in \mathcal{J}}$ such that $\vdash SPR ::_c S$, then according to the proof calculus rule we have that $S = \{UN_i \mapsto S_i\}_{i \in \mathcal{J}}$ and $\vdash SPR_i ::_c S_i$ for $i \in \mathcal{J}$. By the inductive hypothesis we get that $\vdash SPR_i \Rightarrow \mathcal{R}_i$ and $\mathcal{R}_i \models S_i$. Then $\vdash SPR \Rightarrow \mathcal{R}$ where $\mathcal{R} = \{R \mid \text{dom}(R) = \{UN_i\}_{i \in \mathcal{J}}, R(UN_i) \in \mathcal{R}_i \text{ for each } i \in \mathcal{J}\}$ and notice that $\mathcal{R} \models S$ by definition.

Finally, if $SPR = SPR_1 \mathbf{then}\ SPR_2$, with the proof calculus rule we get $\vdash SPR_i ::_c S_i$ for $i = 1, 2$ such that $S = S_1; S_2$ and by induction we have $\vdash SPR_i \Rightarrow \mathcal{R}_i$ and $\mathcal{R}_i \models S_i$, for $i = 1, 2$. It follows (by case analysis, sketched below) using consistency of SPR_1 that $\mathcal{R} = \mathcal{R}_1; \mathcal{R}_2$ is defined and $\mathcal{R} \models S$.

Let us assume that $\vdash SPR_1 \triangleright (U\Sigma_1, U\Sigma_2)$; the other cases are similar. This means that $SPR_2 = (U\Sigma_2, B\Sigma)$. Then $S_1 = (USP_1, USP_2)$ with $\vdash USP_i \triangleright U\Sigma_i$ and $S_2 = (USP'_2, BSP)$ where $\vdash USP'_2 \triangleright U\Sigma_2$ and BSP is a branching specification over $B\Sigma$. Let $(U_2, BM) \in \mathcal{R}_2$. Since $S_1; S_2$ is defined, we have that $USP_2 \rightsquigarrow USP'_2$ and since $\mathcal{R}_1 \models S_1$, there is a unit U_1 of USP_1 such that $(U_1, U_2) \in \mathcal{R}_1$. This gives us an assignment (U_1, BM) and the class of all such assignments is the needed refinement relation \mathcal{R} .

□

Example 11.2.5 We illustrate how the proof calculus for the CASL refinement language applies to the specifications in Ex. 5.1.3.

For the unit specifications **MONOID**, **NATWITHSUC**, **NAT** and **NATBIN**, the refinement specification is of the form (USP, USP) , where USP is in each case the name of the unit specification.

For **R1**, we use the proof calculus rule for simple refinements (second rule in Fig. 11.3) and check that $\mathbf{MONOID} \xrightarrow{\text{Elem} \mapsto \text{Nat}} \mathbf{NAT}$. Since addition of natural num-

bers is associative and 0 is the unit for addition, the refinement condition holds and the refinement R1 is correct. The refinement specification of R1 is (MONOID, NAT).

For R2, we use again the proof calculus rule for simple refinements and check that $\text{NAT} \xrightarrow{\text{Nat} \rightarrow \text{Bin}} \text{NATBIN}$. The refinement specification of R2 is (NAT, NATBIN).

For R3, we need to compose the refinement specifications (MONOID, NAT) and (NAT, NATBIN). The verification condition $\text{NAT} \rightsquigarrow \text{NAT}$ holds trivially and thus the composition of R1 and R2 is correct, with the resulting refinement specification (MONOID, NATBIN).

For ADDITION_FIRST, we first replace the imports with generic units (see Sect. 10.5):

```

arch spec ADDITION_FIRST =
  units N : NAT;
  M : arch spec {
    units F : NAT → NATWITHSUC
    result F[N]}
  result M

```

and the refinement specification of ADDITION_FIRST is $(\text{NATWITHSUC}, \{N \mapsto \text{NAT}, M \mapsto \{F \mapsto \text{NAT} \rightarrow \text{NATWITHSUC}\}\})$.

Correctness of R4 is immediate because the condition $\text{NATWITHSUC} \rightsquigarrow \text{NATWITHSUC}$ holds trivially and R4 has the same refinement specification as ADDITION_FIRST.

Finally for R5, the refinement specification of the anonymous component refinement specification (following **then**) is $\{N \mapsto (\text{NAT}, \text{NATBIN})\}$ and we need to check that it composes with the refinement specification of R4, which we denote by V_4 . Indeed, N is in the domain of the branching specification B on the second component of V_4 and $B(N) = \text{NAT}$; thus, the verification condition $\text{NAT} \rightsquigarrow \text{NAT}$ holds again immediately. The refinement specification of R5 becomes $(\text{NATWITHSUC}, \{N \mapsto \text{NATBIN}, M \mapsto \{F \mapsto \text{NAT} \rightarrow \text{NATWITHSUC}\}\})$.

Example 11.2.6 We now present how the proof calculus applies in the case of the refinements in Ex. 5.1.4.

For REF_SBCS, the second rule in Fig. 11.3 must be applied. This amounts to checking correctness of ARCH_SBCS and proving that

$$\text{SBCS_OPEN} \rightsquigarrow \mathcal{S}_{\text{ARCH_SBCS}}(UE)$$

where $UE = \lambda V : \text{VALUE} \bullet \text{C}[\text{A}[\text{S}[\text{P}[\text{V}]]]]$.

Since ARCH_SBCS contains only unit declarations, each unit is assigned its declared specification and $\mathcal{S}_{\text{ARCH_SBCS}}(UE)$ is obtained as follows:

- UE is a lambda expression, so $\mathcal{S}_{\text{ARCH_SBCS}}(UE) = \text{VALUE} \rightarrow \text{SP}$, where $\text{SP} = \mathcal{S}_{\text{ARCH_SBCS}}(\text{C}[\text{A}[\text{S}[\text{P}[\text{V}]]]]$;

- since $V : \text{VALUE}$ and $P : \text{VALUE} \rightarrow \text{PRELIMINARY}$, the verification condition for the application $P[V]$ is $\text{VALUE} \rightsquigarrow \text{VALUE}$ and holds trivially. The specification of $P[V]$ is PRELIMINARY (because VALUE is included in PRELIMINARY);
- the other units are similar, and the last specification obtained is $\text{STEAM_BOILER_CONTROL_SYSTEM}$.

We obtain thus $\mathcal{S}_{\text{ARCH_SBCS}}(UE) = \text{VALUE} \rightarrow \text{STEAM_BOILER_CONTROL_SYSTEM}$, and since this is precisely SBCS_OPEN , the refinement REF_SBCS is correct. The refinement specification of REF_SBCS is $(\text{SBCS_OPEN}, \text{SPM})$ where $\text{SPM}(P) = \text{VALUE} \rightarrow \text{PRELIMINARY}$, $\text{SPM}(S) = \text{PRELIMINARY} \rightarrow \text{SBCS_STATE}$, $\text{SPM}(A) = \text{SBCS_STATE} \rightarrow \text{SBCS_ANALYSIS}$ and $\text{SPM}(C) = \text{SBCS_ANALYSIS} \rightarrow \text{STEAM_BOILER_CONTROL_SYSTEM}$.

For $\text{REF_SBCS}'$, the third rule in Fig. 11.3 is applied. We have just checked correctness of REF_SBCS and obtained its refinement specification $(\text{SBCS_OPEN}, \text{SPM})$. Therefore, we only have to check correctness of the component refinement following **then** and that the refinement specification obtained, which will be a map SPM' , can be composed to $(\text{SBCS_OPEN}, \text{SPM})$.

With the fourth rule in Fig. 11.3, we must check correctness of the refinement specification of each of the components P , S and A .

For the unit S , we must check correctness of STATE_REF , which amounts to proving that models of SBCS_STATE_IMPL are indeed models of SBCS_STATE . The verification specification obtained is $(\text{STATE_ABSTR}, \text{UNIT_SBCS_STATE})$.

For the unit P , we must check that ARCH_PRELIMINARY is correct. The verification conditions for the two anonymous architectural specifications obtained for MS and MR (as in Sect. 10.5) hold trivially, and we get $\mathcal{S}_{\text{ARCH_PRELIMINARY}}(\text{MS}) = \text{MESSAGES_SENT}$ and $\mathcal{S}_{\text{ARCH_PRELIMINARY}}(\text{MR}) = \text{VALUE} \rightarrow \text{MESSAGES_RECEIVED}$. The specification of the result unit expression (call it UE') of ARCH_PRELIMINARY is of the form $\text{VALUE} \rightarrow \text{SP}'$ where SP' is the specification of the unit term (call it UT) of the lambda expression. By definition, because UT is a unit amalgamation, SP' is the union of the specifications of the terms $\text{SET} [\text{MS fit Elem} \mapsto \text{S_Message}][V]$, $\text{SET} [\text{MR}[V] \text{ fit Elem} \mapsto \text{R_Message}][V]$ and $\text{CST} [V]$ with the specification $\mathcal{S}_{\text{colim}}(\text{UT})$.

For the first term, the verification conditions are

$$\text{MESSAGES_SENT} \models \{\text{sort Elem}\} \text{ with Elem} \mapsto \text{S_Message}$$

and $\text{VALUE} \rightsquigarrow \text{VALUE}$. Both hold immediately and the specification of the term is

$$(\text{SET}[\text{sort Elem}] \text{ with Elem} \mapsto \text{S_Message}) \text{ and } \text{MESSAGES_SENT}$$

For the second term, similarly we obtain the specification

$$(\text{SET}[\text{sort Elem}] \text{ with Elem} \mapsto \text{R_Message}) \text{ and } \text{MESSAGES_RECEIVED}$$

For the third term, notice that $V : \text{VALUE}$ and the condition $\text{VALUE} \rightsquigarrow \text{VALUE}$ holds immediately. Because SBCS_CONSTANTS extends VALUE , the specification obtained is just SBCS_CONSTANTS .

Finally, for the unit A , we need to check correctness of the architectural specification ARCH_ANALYSIS . This does not bring anything new to the cases discussed before and therefore we omit a detailed presentation.

We must then check that the refinement specifications $(\text{SBCS_OPEN}, \text{SPM})$ and SPM' compose. Notice that the domain of SPM' is included in the domain of SPM . For the unit S , the verification condition is immediate and the corresponding specification of S is updated to UNIT_SBCS_STATE . For the unit A , the verification condition holds by noticing that PRELIMINARY is equivalent with the specification SP' obtained in the specification of the result unit of ARCH_PRELIMINARY . SPM is then updated in A to the map taking the units of ARCH_PRELIMINARY to their specification.

11.3 Completeness of the Proof Calculus for Refinements

Recall that the intuition behind Def. 11.2.3 is that we check that a refinement specifies a constructor $\kappa : \text{Mod}(\Sigma) \rightarrow \text{Mod}(\Sigma')$ that takes any model U of some Σ -specification USP to a model $\kappa(U)$ of some Σ' -specification USP' . While we ensure that the constructor should be total on $\text{Mod}(\text{USP})$, the definition does not require that the constructor should be surjective on $\text{Mod}(\text{USP}')$. In particular, there can be another Σ' -specification USP'' that provides a more precise description of $\kappa(\text{Mod}(\text{USP}))$. This is obvious in the case of simple refinements: if the refinement $\text{R} = \text{USP}$ **refined via** σ **to** USP' is correct, we have that $\vdash \text{R} \Rightarrow \mathcal{R}$ where $\mathcal{R} = \{(u|_{\sigma}, u) \mid u \in \text{Unit}(\text{USP}')\}$. Then the models produced by the constructor associated with R are not necessarily all USP -models, but generally only $(\text{USP}' \text{ hide } \sigma)$ -models.

Architectural specification defines a construction that appears in a top-down development process, when a given requirement specification is implemented by a number of specifications and these specifications act like an interface between the components of the architectural specification. In particular, this introduces an abstraction barrier between a given requirement specification and its architectural refinement. The implementation of the units of an architectural specification can be changed (as long as they respect their specifications) while keeping the possibility to assemble the units to a system satisfying the given requirement specification.

The calculus that we introduced in the previous section respects this abstraction barrier. However, this comes at a certain price: Ex. 11.3.1 below shows that in some cases, we can prove a refinement correct only if we exploit the properties induced by a certain implementation we have chosen during development of the system. Using information about the choice of implementation gives a more precise description of the image of a constructor. But this of course breaks the above mentioned abstraction barrier. In our case, we have to make use of a particular choice for a refinement of such a component when proving correctness of the entire development.

Hence, we cannot expect to prove completeness of the calculus introduced above, just because it respects the abstraction barrier. In the following we will

introduce an enhanced proof calculus that can be proven complete and hence necessarily breaks the abstraction barrier.

Example 11.3.1 *Let us consider the following specifications:*

spec SIG = **sort** s
 ops $x, y : s$

spec EQ = SIG **then** $\{ \bullet x = y \}$

refinement INCL = SIG **refined to** EQ

arch spec ASP = **units** M : SIG **result** M

refinement ASP_EQ = **arch spec** ASP **then** $\{M \text{ to INCL}\}$

refinement REF_EQ = EQ **refined to** ASP_EQ

Here the tree of ASP (in the sense of Sect. 11.4) “grows” in both directions: first the branch corresponding to the unit M is extended via the refinement ASP_EQ, then the tree of ASP_EQ “grows” towards the root, via the refinement REF_EQ.

We have that $\vdash \text{ASP} ::_c (\text{SIG}, \{M \mapsto \text{SIG}\})$ and $\vdash \{M \text{ to INCL}\} ::_c \{M \mapsto (\text{SIG}, \text{EQ})\}$. Thus, $\vdash \text{ASP_EQ} ::_c (\text{SIG}, \{M \mapsto \text{EQ}\})$ and the verification condition of REF_EQ is $\text{EQ} \rightsquigarrow \text{SIG}$, which obviously does not hold. This is due to the fact that we do not make use of the fact that M has been further refined when proving the correctness of the entire development: in the specification of ASP_EQ we have only modified the specification of the component M. However, this induces a restriction on the domain of the associated constructor, and the codomain gets restricted as well. This change is not captured in the definition of the compositions of refinement specifications.

We can simplify the task of keeping track of the changes by expressing every refinement as equivalent architectural specification(s), in the sense that their models are in a one-to-one correspondence, as we will see below. We have seen that for any architectural specification ASP, \mathcal{S}_{ASP} captures exactly (under some conditions) the models produced by the architectural specification. The idea is that rather than defining \mathcal{S}_{ASP} as a specification, we define it as a *specification expression* involving the specifications of the units involved. The expression is then evaluated when proving a refinement and composition of refinements induces substitution of the specification of the refined unit with the expression associated to the refinement we compose with. This gives a dynamic nature to the verification process: at each moment, the specifications provide only a snapshot of the model classes involved, and they can be further restricted by compositions.

Example 11.3.2 *Let us again assume that the refinement $R = \text{USP}$ refined via σ to USP' is correct, and then we have that $\vdash R \Rightarrow \mathcal{R}$ where $\mathcal{R} = \{(U|_\sigma, U) \mid U \in \text{Unit}(\text{USP}')\}$.*

The same constructor as the one specified by R , namely $\mathbf{Mod}(\sigma) : \mathit{Unit}(USP') \rightarrow (\mathit{Unit}(USP') \mathbf{hide} \sigma)$, can be equivalently specified by the architectural specification

arch spec $ASP =$
units $UN : USP'$
result $UN \mathbf{hide} \sigma$

Given that $\vdash ASP \Rightarrow \{(U|_\sigma, \{UN \mapsto U\}) \mid U \in \mathit{Unit}(USP')\}$, we get a one-to-one correspondence between the assignment $(U|_\sigma, U)$ in the model class of R , on one side, and $(U|_\sigma, \{UN \mapsto U\})$ in the model class of ASP , on the other side, for each $U \in \mathit{Unit}(USP')$. In the general case, when such a correspondence can be set, we will say that the models of R and ASP correspond up to unit names.

$$\begin{array}{c}
 \frac{\mathbf{arch\ spec\ } ASP = \mathbf{units\ } UN : USP' \mathbf{result\ } UN}{\vdash USP ::_e ASP} \\
 \\
 \frac{\vdash SPR' ::_e ASP' \quad USP \overset{\sigma}{\rightsquigarrow} \mathcal{S}_{ASP'}}{\mathbf{arch\ spec\ } ASP = \mathbf{units\ } UN : \mathbf{arch\ spec\ } ASP' \mathbf{result\ } (UN \mathbf{hide} \sigma)} \\
 \frac{}{\vdash USP \mathbf{refined\ via\ } \sigma \mathbf{to\ } SPR' ::_e ASP} \\
 \\
 \frac{\vdash SPR_i ::_e ASP_i}{\mathbf{arch\ spec\ } ASP' = ASP[\{UN_i / \mathbf{arch\ spec\ } ASP_i\}_{i=1, \dots, n}]} \\
 \frac{}{\vdash ASP = \mathbf{units\ } \{UN_i : SPR_i\}_{i=1, \dots, n} \mathbf{result\ } UE ::_e ASP'} \\
 \\
 \frac{\vdash SPR_i ::_e \mathcal{VS}_i}{\vdash \{UN_i \mathbf{to\ } SPR_i\}_{i \in \mathcal{J}} ::_e \{UN_i \mathbf{to\ } \mathcal{VS}_i\}_{i \in \mathcal{J}}} \\
 \\
 \frac{\vdash SPR_1 ::_e \mathcal{VS}_1 \quad \vdash SPR_2 ::_e \mathcal{VS}_2 \quad \mathcal{VS} = \mathcal{VS}_1; \mathcal{VS}_2}{\vdash SPR_1 \mathbf{then\ } SPR_2 ::_e \mathcal{VS}}
 \end{array}$$

Figure 11.4: Enhanced calculus for refinements.

Note that in the case of component refinements, the refinement of each component needs to be replaced by an equivalent architectural specification. We define then *enhanced verification specifications* \mathcal{VS} :

$$\mathcal{VS} ::= ASP \mid \{UN_i \mapsto \mathcal{VS}_i\}_{i \in \mathcal{J}}$$

where ASP is an architectural specification and we call enhanced verification specifications of the form $\{UN_i \mapsto \mathcal{VS}_i\}_{i \in \mathcal{J}}$ enhanced verification maps. The idea is

that if \mathcal{VS} is the enhanced verification specification of a refinement SPR , \mathcal{VS} specifies the constructor(s) induced by SPR . Moreover, we define a partial operation of *composition* between enhanced verification specifications, denoted $\mathcal{VS}_1; \mathcal{VS}_2$ as follows:

- if $\mathcal{VS}_1 = ASP_1$ such that ASP_1 has no branching² and \mathcal{VS}_2 is also an architectural specification ASP_2 , then $\mathcal{VS}_1; \mathcal{VS}_2$ replaces the specification of the leaf in ASP_1 with ASP_2 ;
- if $\mathcal{VS}_1 = ASP_1$ and \mathcal{VS}_2 is an enhanced verification map, then $\mathcal{VS}_1; \mathcal{VS}_2$ is defined iff for each $UN \in \text{dom}(\mathcal{VS}_2)$, UN is in $\text{dom}(\mathcal{VS}_1)$ such that
 - if the specification of UN in ASP_1 is a unit specification USP , then $\mathcal{VS}_2(UN)$ must be an architectural specification ASP_2 and $USP \rightsquigarrow S_{ASP_2}$. Then we update the specification of UN in ASP_1 to ASP_2 ;
 - if the specification of UN in ASP_1 is an architectural specification ASP' , then $\mathcal{VS}_2(UN)$ must be an enhanced verification map \mathcal{VS}' such that $ASP'; \mathcal{VS}'$ is defined. Then we update the specification of UN in ASP to $ASP'; \mathcal{VS}'$;
- if \mathcal{VS}_1 is an enhanced verification map, then \mathcal{VS}_2 must be an enhanced verification map as well and we define $\mathcal{VS}_1; \mathcal{VS}_2$ by modifying the ill-defined union of \mathcal{VS}_1 and \mathcal{VS}_2 , putting $\mathcal{VS}_1; \mathcal{VS}_2(UN) = \mathcal{VS}_1(UN); \mathcal{VS}_2(UN)$ for each unit name UN in the intersection of the domains of \mathcal{VS}_1 and \mathcal{VS}_2 .

Models of enhanced verification specifications are obvious generalizations of models of architectural specifications: if a verification specification \mathcal{VS} is an architectural specification ASP , then the model of \mathcal{VS} is just the architectural model \mathcal{AM} such that $\vdash ASP \Rightarrow \mathcal{AM}$, while if \mathcal{VS} is an enhanced verification map $\{UN_i \mapsto \mathcal{VS}_i\}_{i \in \mathcal{J}}$ and \mathcal{M}_i is a model for \mathcal{VS}_i , for each $i \in \mathcal{J}$, then $\{UN_i \mapsto \mathcal{M}_i\}_{i \in \mathcal{J}}$ is a model for \mathcal{VS} . By a slight abuse of notation, we denote the models of verification specifications also with \mathcal{AM} , as in the case of architectural specifications, and we write $\vdash \mathcal{VS} \Rightarrow \mathcal{AM}$ to denote that \mathcal{AM} is the model of \mathcal{VS} .

An enhanced proof calculus for refinements is presented in Fig. 11.4, with judgements of the form $\vdash SPR ::_e \mathcal{VS}$, where SPR is a refinement and \mathcal{VS} is an enhanced verification specification. In the rule for architectural specifications, we denote by $ASP[\{UN_i/\mathbf{arch\ spec}\ }_{i=1, \dots, n}]$ the architectural specification obtained from an architectural specification ASP with unit declarations $UN_i : SPR_i, i = 1, \dots, n$ by replacing the specification of each declared unit UN_i with some architectural specification ASP_i and keeping all unit definitions as in ASP .

²This means that the architectural specification has just one unit, and the specification of that unit can be architectural only if it has one component itself, and so on, until a leaf is reached when the specification of the current unit is a unit specification.

Theorem 11.3.3 (Soundness) *Let SPR be a refinement such that $\vdash SPR \triangleright \square$ and all generic units in the architectural specifications appearing in SPR are consistent. If $\vdash SPR ::_e \mathcal{VS}$ for some enhanced verification specification \mathcal{VS} , then $\vdash SPR \Rightarrow \mathcal{R}$ for some refinement relation \mathcal{R} , and $\vdash \mathcal{VS} \Rightarrow \mathcal{AM}$ for some \mathcal{AM} such that \mathcal{R} and \mathcal{AM} correspond up to unit names.*

Proof. Induction on the structure of SPR .

If $SPR = USP$, then $\vdash USP ::_e ASP$ where **arch spec** $ASP = \mathbf{units} \ UN : USP \mathbf{result} \ UN$. By definition, $\vdash USP \Rightarrow \mathcal{R}$ where $\mathcal{R} = \{(U, U) \mid U \in \mathit{Unit}(USP)\}$. With the model semantics rules for architectural specifications, we know that $\vdash ASP \Rightarrow \mathcal{AM}$, where $\mathcal{AM} = \{(U, \{UN \mapsto U\}) \mid U \in \mathit{Unit}(USP)\}$. It is obvious that \mathcal{AM} and \mathcal{R} correspond up to unit names.

If $SPR = USP$ refined via σ to SPR' and by the hypothesis $\vdash SPR ::_e ASP$, then according to the proof calculus we know that $\vdash SPR' ::_e ASP'$, where ASP' is as in the rule for simple refinements. By the induction hypothesis we get that $\vdash SPR' \Rightarrow \mathcal{R}'$ and \mathcal{R}' and \mathcal{AM}' correspond up to unit names, where $\vdash ASP' \Rightarrow \mathcal{AM}'$. Moreover, the refinement condition $USP \overset{\sigma}{\rightsquigarrow} \mathcal{S}_{ASP'}$ holds. Let $(U, BM) \in \mathcal{R}'$ and since \mathcal{R}' and \mathcal{AM}' correspond up to unit names we get that $U \models \mathcal{S}_{ASP'}$ and therefore $U|_{\sigma} \models USP$. Then by definition we have that $\vdash SPR \Rightarrow \mathcal{R}$, where $\mathcal{R} = \{(U|_{\sigma}, BM) \mid (U, BM) \in \mathcal{R}'\}$. With the model semantics rules for architectural specifications, we get that $\vdash ASP \Rightarrow \mathcal{AM}$ where $\mathcal{AM} = \{(U|_{\sigma}, BM) \mid (U, BM) \in \mathcal{AM}'\}$ and since \mathcal{R}' and \mathcal{AM}' correspond up to unit names, so do \mathcal{R} and \mathcal{AM} .

If $SPR = \mathbf{arch\ spec} \ ASP$, then by the hypothesis for any unit declaration $UN_i : SPR_i$ in ASP , $\vdash SPR_i ::_e ASP_i$. By the induction hypothesis this means $\vdash SPR_i \Rightarrow \mathcal{R}_i$ and \mathcal{R}_i corresponds up to unit names with the model class \mathcal{AM}_i of ASP_i . This means that for any $(U, BM) \in \mathcal{R}_i$, $U \models \mathcal{S}_{ASP_i}$. Then for any choice of models $(U_i, BM_i) \in \mathcal{R}_i$ for each i , we obtain on one hand a unit environment for ASP by projecting the models on the first component and we can combine them according to the result unit expression of ASP to obtain a model U . On the other hand, with the same construction we get a unit environment for ASP' (where ASP' is the architectural specification defined in the corresponding proof rule of the calculus) and since the result unit expression is the same, we get the same model U . Then the models of ASP and ASP' correspond up to unit names.

If $SPR = \{UN_i \mathbf{to} \ SPR_i\}_{i \in \mathcal{J}}$, then by the hypothesis we know $\vdash SPR_i ::_e \mathcal{VS}_i$ and by the inductive hypothesis we have $\vdash SPR_i \Rightarrow \mathcal{R}_i$ and \mathcal{R}_i corresponds with the model class of \mathcal{VS}_i . Then using the model semantics rule we get $\vdash SPR \Rightarrow \mathcal{R} = \{UN_i \rightarrow \mathcal{R}_i\}_{i \in \mathcal{J}}$ and \mathcal{R} corresponds with the model class of $\{UN_i \rightarrow \mathcal{VS}_i\}_{i \in \mathcal{J}}$.

Finally, if $SPR = SPR_1 \mathbf{then} \ SPR_2$ and $\vdash SPR ::_c \mathcal{VS}$, then by the corresponding proof rule $\vdash SPR_i ::_c \mathcal{VS}_i$ for $i = 1, 2$ and $\mathcal{VS} = \mathcal{VS}_1; \mathcal{VS}_2$. By the induction hypothesis we get $\vdash SPR_i \Rightarrow \mathcal{R}_i$ and \mathcal{R}_i corresponds to the model class of \mathcal{VS}_i , for $i = 1, 2$. We show that $\mathcal{R}_1; \mathcal{R}_2$ is defined and it corresponds to the model class of \mathcal{VS} by case analysis on the refinement signature $R\Sigma_1$ of SPR_1 .

If $R\Sigma_1 = (U\Sigma, U\Sigma')$ then $R\Sigma_2 = (U\Sigma', B\Sigma)$. Then, \mathcal{VS}_1 and \mathcal{VS}_2 are both architectural specifications and \mathcal{VS}_1 has only one unit UN labelled with a unit specifica-

tion USP (possibly inside several architectural levels). Therefore the assignments in \mathcal{R}_2 are branching assignments of the form (U, BM) and by induction we know that $U \models \mathcal{S}_{\mathcal{V}\mathcal{S}_2}$. Since $\mathcal{V}\mathcal{S}_1; \mathcal{V}\mathcal{S}_2$ is defined, this implies that $U \models USP$ and there must be an assignment (U_0, U) of \mathcal{R}_1 , because $\mathcal{V}\mathcal{S}_1$ is architectural and therefore U generates a unit environment for $\mathcal{V}\mathcal{S}_1$ in which the result unit of $\mathcal{V}\mathcal{S}_1$ is evaluated to a model U_0 , and \mathcal{AM}_1 and \mathcal{R}_1 correspond up to unit names. This means that $\mathcal{R}_1; \mathcal{R}_2$ is defined. Moreover, if we replace USP in $\mathcal{V}\mathcal{S}_1$ with the result of evaluating $\mathcal{S}_{\mathcal{V}\mathcal{S}_2}$ in the context given by $\mathcal{V}\mathcal{S}_2$, we get that the model U_0 satisfies $\mathcal{S}_{\mathcal{V}\mathcal{S}_1}[UN/\mathcal{S}_{\mathcal{V}\mathcal{S}_2}]$, which ensures that $\mathcal{R}_1; \mathcal{R}_2$ corresponds to the model class of $\mathcal{V}\mathcal{S}_1; \mathcal{V}\mathcal{S}_2$. The other cases are similar but tedious and we omit them. \square

Theorem 11.3.4 (Completeness) *Let SPR be a refinement such that $\vdash SPR \triangleright \square$ and all the generic units in the architectural specifications appearing in SPR are consistent and applied only once. If $\vdash SPR \Rightarrow \mathcal{R}$ then $\vdash SPR ::_e \mathcal{V}\mathcal{S}$ for some enhanced verification specification $\mathcal{V}\mathcal{S}$ and $\vdash \mathcal{V}\mathcal{S} \Rightarrow \mathcal{AM}$ for some \mathcal{AM} such that \mathcal{R} and \mathcal{AM} correspond up to unit names.*

Proof. Induction on the structure of SPR .

If $SPR = USP$, then $\vdash SPR ::_e ASP$ and $\vdash A(SPR) \Rightarrow \mathcal{AM}$, where $\mathcal{AM} = \{(U, \{UN \mapsto U\}) \mid U \in \text{Unit}(USP)\}$. By definition, $\vdash USP \Rightarrow \mathcal{R}$ where $\mathcal{R} = \{(U, U) \mid U \in \text{Unit}(USP)\}$ and therefore the one-to-one correspondence is obvious.

If $SPR = USP$ refined via σ to SPR' and we know that $\vdash SPR \Rightarrow \mathcal{R}$, then with the model semantics rules we get that $\vdash SPR' \Rightarrow \mathcal{R}'$ and for every $(U, BM) \in \mathcal{R}'$, $U|_\sigma \in \text{Unit}(USP)$. By the inductive hypothesis we have that $\vdash SPR' ::_e ASP'$ and there is a one-to-one correspondence between \mathcal{R}' and \mathcal{AM}' , where $\vdash ASP' \Rightarrow \mathcal{AM}'$. We need to prove that $USP \overset{\sigma}{\rightsquigarrow} \mathcal{S}_{ASP'}$, so let $V \models \mathcal{S}_{ASP'}$. We know that there is a branching model BM such that $(V, BM) \in \mathcal{AM}'$ and we know that (V, BM) corresponds with an assignment of \mathcal{R}' , which gives us that $V|_\sigma \models USP$. Thus $\vdash SPR ::_e ASP$. Finally, notice that the model class of ASP and \mathcal{R} correspond by construction of ASP , definition of \mathcal{R} and correspondence between \mathcal{AM}' and \mathcal{R}' .

If $SPR = \text{arch spec } ASP$ and $\vdash ASP \Rightarrow \mathcal{R}$, then by the model semantics rule we get that $\vdash SPR_i \Rightarrow \mathcal{R}_i$ for each unit declaration $UN_i : SPR_i$ in ASP . By induction we get that $\vdash SPR_i ::_e ASP_i$ and model class of ASP_i correspond with model class of \mathcal{R}_i . It follows that $\vdash SPR ::_e ASP'$ and the model of ASP' corresponds with the model class of \mathcal{R} with the same argument as in the soundness proof.

If $SPR = \{UN_i \text{ to } SPR_i\}_{i \in \mathcal{J}}$ and $\vdash SPR \Rightarrow \mathcal{R}$, then with the model semantics rule we get $\vdash SPR_i \Rightarrow \mathcal{R}_i$. By induction we get $\vdash \mathcal{R}_i ::_e \mathcal{V}\mathcal{S}_i$ and with the proof calculus rule we get $\vdash \mathcal{R} ::_e \mathcal{V}\mathcal{S} = \{UN_i \text{ to } \mathcal{V}\mathcal{S}_i\}_{i \in \mathcal{J}}$. The correspondence between \mathcal{R} and the model of $\mathcal{V}\mathcal{S}$ follows immediately by definition and the correspondence between \mathcal{R}_i and the model of $\mathcal{V}\mathcal{S}_i$, for each $i \in \mathcal{J}$.

Finally, if $SPR = SPR_1$ then SPR_2 , and $\vdash SPR \Rightarrow \mathcal{R}$. then according to the model semantics rule we get that $\vdash SPR_i \Rightarrow \mathcal{R}_i$ for $i = 1, 2$ and $\mathcal{R} = \mathcal{R}_1; \mathcal{R}_2$. By induction we get that $\vdash SPR_i ::_e \mathcal{VS}_i$ and \mathcal{R}_i corresponds with the model class of \mathcal{VS}_i for $i = 1, 2$. We need show that $\mathcal{VS} = \mathcal{VS}_1; \mathcal{VS}_2$ is defined.

We will consider only the case when \mathcal{VS}_1 is an architectural specification with more than a component; the other cases are similar. We know that \mathcal{VS}_2 is an enhanced verification map since the refinements compose. Let $UN \in dom(\mathcal{VS}_2)$ and assume the specification of UN in \mathcal{VS}_1 is a unit specification USP (the case when the specification of U is architectural reduces to this case) and $\mathcal{VS}_2(UN) = ASP'$. Let $U \models S_{ASP'}$ and then by Thm. 11.1.12 we know that there is a BM such that (U, BM) is an architectural model of ASP' . Since the models of \mathcal{VS}_2 correspond with \mathcal{R}_2 and $\mathcal{R}_1; \mathcal{R}_2$ is defined, there must be an assignment (U', BM') in \mathcal{R}_1 such that $BM'(UN) = U$. We know that (U, BM) corresponds with a \mathcal{VS}_1 -model, which implies $U \models USP$, so the refinement condition holds. To prove the correspondence between \mathcal{R} and architectural models of \mathcal{VS} , let $(U, BM) \in \mathcal{VS}$. Notice that by construction of \mathcal{VS} , BM does not only produce a unit environment for \mathcal{VS} but also a unit environment for \mathcal{VS}_1 such that evaluating the result unit expression of \mathcal{VS} and \mathcal{VS}_1 yields the same result. This shows that we can write (U, BM) as a composition between a model corresponding to an assignment in \mathcal{R}_1 and a model corresponding to an assignment in \mathcal{R}_2 .

□

11.4 Refinement Trees

We now give a formal definition of the concept of *refinement tree*. Refinement trees provide visualization means for the structure of the development and access points in HETS to the logical properties of architectural specifications and refinements. Refinement trees complement development graphs, that represent the structure of the specifications involved and that can be used for discharging simple refinement proof obligations. The refinement structure, as captured by refinement trees, may be orthogonal to the specification structure.

While intuitively clear, refinement trees have a slightly involved formalization. This is because they are built in a stepwise manner and must be *combined* in the way prescribed by the refinements: composition of refinements gives rise to composition of refinement trees, and we need a mechanism for keeping track of the branches and nodes to retrieve the appropriate connection points between trees. Moreover, in the case of component refinement, each component produces a (sub)tree.

Example 11.4.1 Fig. 11.5 presents the refinement tree of the specifications in Ex. 5.1.4. Single arrows denote components, while double arrows denote refinements. Notice that in the case of e.g. REF_SBCS", we need to build the trees of the architectural specifications ARCH_FAILURE_DETECTION and ARCH_PREDICTION, store them as corresponding to the units FD and PR in a component refinement, obtaining thus

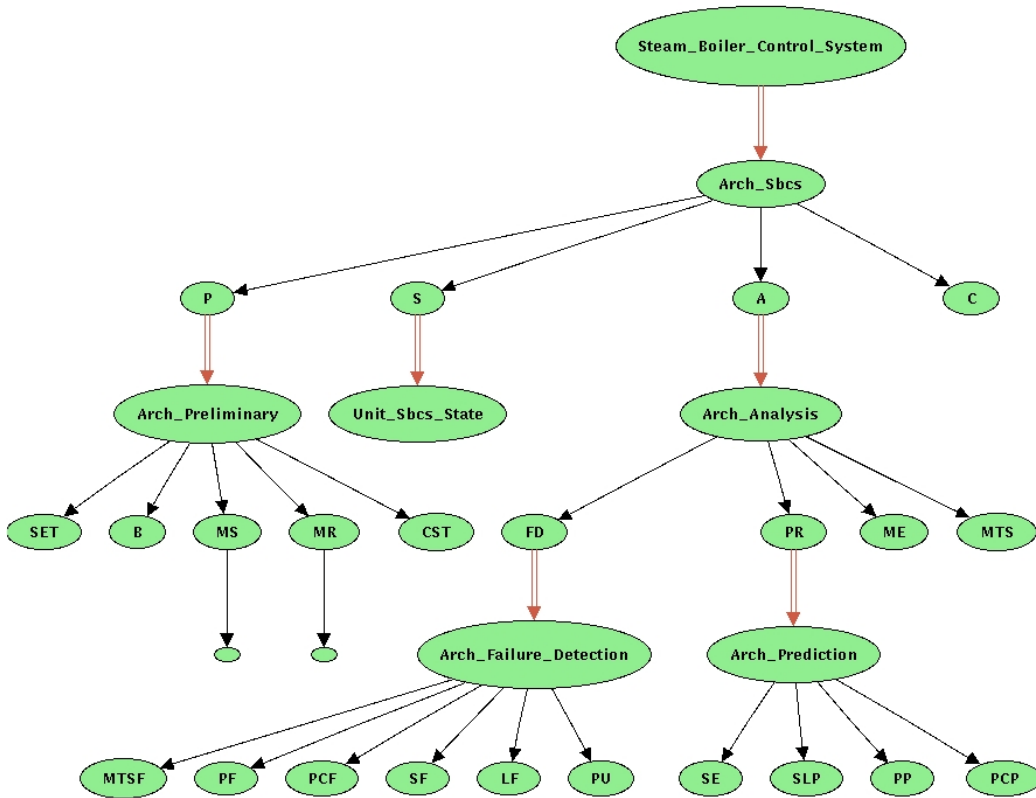


Figure 11.5: The refinement tree of the steam boiler control system.

a set of trees. We must then connect these trees to the corresponding components of the unit A, and thus we must be able to identify their nodes in the refinement tree of ARCH_ANALYSIS by name.

The example shows that refinement trees should consist of a collection of trees and that they can grow not only at the leaves, but also at the root, thus old roots becoming subtrees. This leads to the following definition.

Definition 11.4.2 A refinement tree \mathcal{RT} consists of a set of trees, where a tree has:

- nodes labelled with unit specifications and
- directed edges between nodes n_1, n_2 of the tree, that can be either
 1. refinement links $n_1 \Rightarrow n_2$ (to denote refinement of specifications) or
 2. component links $n_1 \rightarrow n_2$, (to denote architectural decomposition).

We need to define an auxiliary structure to keep track of the roots, leaves and nodes of the branching decompositions; this will make it possible to compose refinement trees. We would like to stress that this information is only needed for book-keeping during the construction of refinement trees, and it can be dispensed with when looking at a completed refinement tree. Let us define *refinement tree pointers* in a refinement tree \mathcal{RT} as either (i) *simple refinement pointers* of the form (n_1, n_2) with n_1, n_2 nodes in \mathcal{RT} , with the intuition that the first node is the root and the second node is the leaf of a path through the tree³, (ii) *branching refinement pointers* of the form (n, f) , where n is a node and f is a map assigning refinement tree pointers to unit names, or (iii) *component refinement pointers* which are maps assigning refinement tree pointers to unit names.

Let us give a series of notations and operations on refinement trees. We denote \mathcal{RT}_\emptyset the empty tree. If \mathcal{RT} is a refinement tree, $\mathcal{RT}[USP]$ is obtained by adding to it a new isolated node n labelled with *USP*. For $\mathcal{RT}_1, \dots, \mathcal{RT}_k$ refinement trees, $\mathcal{RT}[n_1 \rightarrow \mathcal{RT}_1, \dots, \mathcal{RT}_k]$ denotes the tree obtained by inserting component links from the node n_1 of \mathcal{RT} to the roots of each of the argument trees. Moreover, refinement trees can be composed as defined below.

Definition 11.4.3 Given two refinement trees \mathcal{RT}_1 with pointer p_1 and \mathcal{RT}_2 with pointer p_2 we denote (\mathcal{RT}, p) the composition $\mathcal{RT}_1 \circ_{p_1, p_2} \mathcal{RT}_2$ defined as follows:

1. if p_1 is a simple refinement pointer (n_1, n_2) and p_2 is a simple refinement pointer (m_1, m_2) , \mathcal{RT} is obtained by adding a refinement link from n_2 to the successor of the node m_1 along the path (m_1, m_2) in \mathcal{RT}_2 . The pointer p is then (n_1, m_2) .
2. if p_1 is a simple refinement pointer (n_1, n_2) and p_2 is a branching refinement pointer (m_1, f) , \mathcal{RT} is obtained by adding a refinement link from n_2 to m_1 . The pointer p is (n_1, f) .
3. if p_1 is a branching refinement pointer (n_1, f_1) and p_2 is a component refinement pointer f_2 , \mathcal{RT} is obtained by making for each X in $\text{dom}(f_2)$ the composition of the subtree pointed by $f_1(X)$ with the tree pointed by $f_2(X)$, which also returns a pointer p_X for each X in $\text{dom}(f_2)$. The pointer p is $(n_1, f_1[f_2])$, where $f_1[f_2]$ updates the value of X in f_1 with the pointer p_X .
4. if p_1 is a component refinement pointer f_1 and p_2 is a component refinement pointer p_2 , then \mathcal{RT} is obtained by making for each X in $\text{dom}(f_2)$ the composition of the subtree pointed by $f_1(X)$ with the tree pointed by $f_2(X)$, which also returns a pointer p_X for each X in $\text{dom}(f_2)$. The pointer p is $f_1[f_2]$.

The composition is undefined otherwise.

The refinement trees will be constructed for correct refinements, in parallel with the verification process. To ease understanding, we have separated the parts that build the refinement trees, as presented in Fig. 11.6.

³Notice that the two nodes can coincide.

$$\begin{array}{c}
\frac{(n, \mathcal{RT}) = \mathcal{RT}_\emptyset[USP]}{\vdash USP ::_c \mathcal{RT}, (n, n)} \quad \frac{\begin{array}{l} \vdash USP ::_c \mathcal{RT}_1, p_1 \\ \vdash SPR ::_c \mathcal{RT}_2, p_2 \\ (\mathcal{RT}, p) = \mathcal{RT}_1 \circ_{p_1, p_2} \mathcal{RT}_2 \end{array}}{\vdash USP \text{ refined via } \sigma \text{ to } SPR ::_c \mathcal{RT}, p} \\
\\
\frac{\begin{array}{l} \vdash SPR_i ::_c \mathcal{RT}_i, p_i \\ UE \text{ is the result unit of } ASP \\ (n, \mathcal{RT}') = \mathcal{RT}_\emptyset[\mathcal{S}_\Gamma(UE)] \\ \mathcal{RT} = \mathcal{RT}'[n \rightarrow \mathcal{RT}_1, \dots, \mathcal{RT}_k] \\ p = (n, \{UN_i \rightarrow p_i\}_{i=1, \dots, k}) \end{array}}{\vdash ASP ::_c \mathcal{RT}, p} \quad \frac{\begin{array}{l} \text{for each } i \in \mathcal{J} \\ \vdash SPR_i ::_c \mathcal{RT}_i, p_i \\ \mathcal{RT} = \cup_{i \in \mathcal{J}} \mathcal{RT}_i \\ p = \{UN_i \mapsto p_i\}_{i \in \mathcal{J}} \end{array}}{\vdash \{UN_i \text{ to } SPR_i\}_{i \in \mathcal{J}} ::_c \mathcal{RT}, p} \\
\\
\frac{\begin{array}{l} \vdash SPR_1 ::_c \mathcal{RT}_1, p_1 \\ \vdash SPR_2 ::_c \mathcal{RT}_2, p_2 \\ (\mathcal{RT}, p) = \mathcal{RT}_1 \circ_{p_1, p_2} \mathcal{RT}_2 \end{array}}{\vdash SPR_1 \text{ then } SPR_2 ::_c \mathcal{RT}, p}
\end{array}$$

Figure 11.6: Construction of refinement trees.

Example 11.4.4 We can illustrate the definition of composition of refinement trees with the help of Ex. 5.1.3. We will use the names of the specifications and of the units of architectural specifications to identify nodes in the refinement trees and therefore the names will also appear in pointers. The refinement trees and the results of their compositions are presented in Fig. 11.7.

Firstly, the refinements R1 and R2 are composed in R3 to form a chain; this is case (1) of the definition of composition of refinement trees. The pointer p_1 of R1 is $(Monoid, Nat)$ and the pointer p_2 of R2 is $(Nat, NatBin)$. The pointer of the result of their composition is $(Monoid, NatBin)$.

The refinement tree of R4 is obtained by composing the tree with a single node and pointer $q_1 = (NatWithSuc, NatWithSuc)$ with the tree of the architectural specification ADDITION_FIRST (case (2)). The pointer of the latter is $q_2 = (Addition_First, \{M \mapsto (M, M), N \mapsto (N, N)\})$. After composition, the pointer q_3 is $(NatWithSuc, \{M \mapsto (M, M), N \mapsto (N, N)\})$.

Finally, the tree of R5 is obtained by composing the tree obtained at the previous step, of pointer q_3 , with the set containing the tree of R2 with pointer $q_4 = \{N \mapsto (Nat, NatBin)\}$. The pointer of the result is $(NatWithSuc, \{M \mapsto (M, M), N \mapsto (N, NatBin)\})$ - case (3).

We can obtain an example for case (4) by writing the refinement of the components equivalently, but in a different way:

refinement R = {

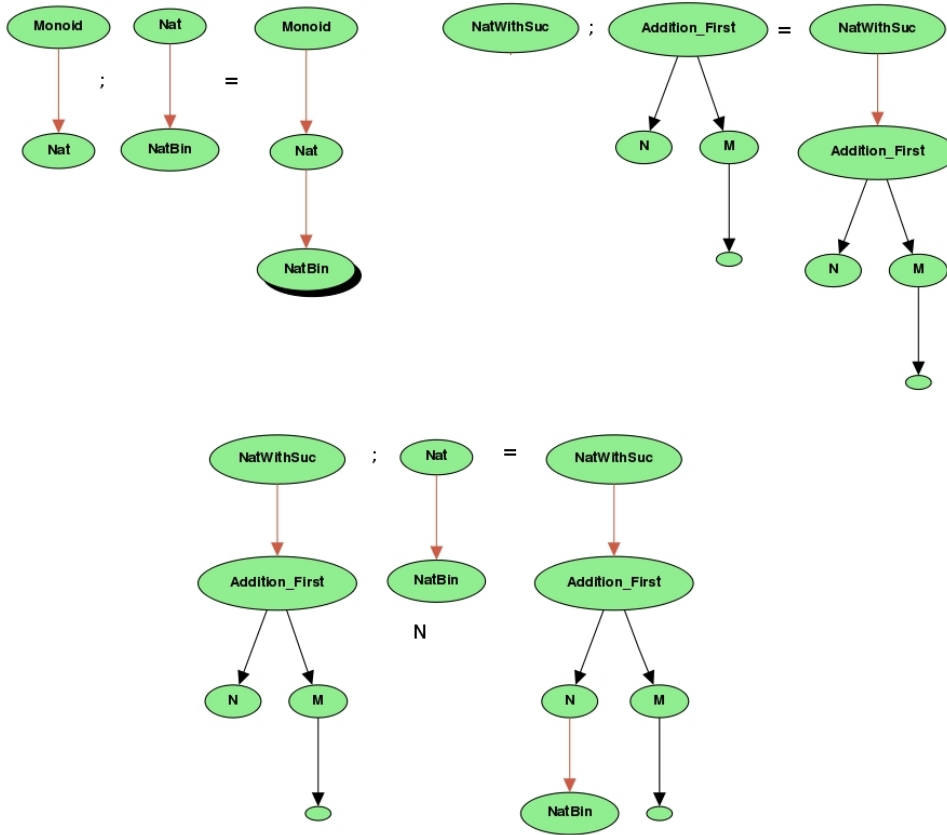


Figure 11.7: Composition of refinement trees.

P to *arch spec* ARCH_PRELIMINARY, *S* to STATEREF,
A to *arch spec* ARCH_ANALYSIS }
 then {
A to {FD to *arch spec* ARCH_FAILURE_DETECTION,
 PR to *arch spec* ARCH_PREDICTION } }

and the corresponding refinement trees are obtained by removing the first three levels from the tree in Fig. 11.5.

11.5 Checking Consistency of Refinement Specifications

We introduce a calculus for checking whether a refinement specification is consistent, i.e. it has a refinement model. [Kutz and Mossakowski, 2011] successfully apply this calculus to verify the consistency of the upper ontology Dolce. Dolce is too large for contemporary model finders. Instead of hand-crafting a large and specific model, the consistency of Dolce is shown using an architectural refinement. This has the advantage of giving a modular model for Dolce, i.e. one that can be

$\frac{\vdash \text{cons}(USP)}{\vdash \text{cons}(USP \text{ qua SPEC-REF})}$	$\frac{\vdash \text{cons}(SPR)}{\vdash \text{cons}(USP \text{ refined via } \sigma \text{ to } SPR)}$
$\frac{\vdash \text{cons}(SPR) \text{ for each } UN : SPR \text{ in } ASP}{\vdash \text{cons}(ASP)}$	$\frac{\vdash \text{cons}(SPR_i), i \in \mathcal{J}}{\vdash \text{cons}(\{UN_i \text{ to } SPR_i\}_{i \in \mathcal{J}})}$
$\frac{\begin{array}{l} SPR_1 \text{ contains branchings} \\ \vdash \text{cons}(SPR_1) \\ \vdash \text{cons}(SPR_2) \end{array}}{\vdash \text{cons}(SPR_1 \text{ then } SPR_2)}$	$\frac{\begin{array}{l} SPR_1 \text{ does not contain branchings} \\ \vdash \text{cons}(SPR_2) \end{array}}{\vdash \text{cons}(SPR_1 \text{ then } SPR_2)}$

Figure 11.8: Consistency calculus for refinements.

changed at various local places (= leaves of the refinement tree) without affecting the possibility to assemble (via the semantics of architectural specifications) a global model of Dolce.

The proof calculus for refinements relies on an obvious observation made already in [Sannella and Tarlecki, 1988a] that constructors preserve consistency. Intuitively, a refinement is consistent if its target is, and an architectural specification is consistent if all its unit specifications are. This makes it clear that our calculus eventually (for checking consistency of the leaves of the refinement tree) must be based on a calculus for the consistency of unit specifications, which we denote $\vdash \text{cons}(USP)$ and is given by the rules in Fig. 11.8. Checking consistency of non-parametric unit specification amounts to checking consistency of structured specifications; a calculus for this has been introduced in [Roggenbach and Schröder, 2001] (this is in turn based on some institution-specific calculus for consistency of basic specifications). Checking consistency of parametric unit specification amounts to checking conservativity of extensions of structured specifications; for the case of first-order logic and CASL basic specifications, a sound but necessarily incomplete calculus has been developed in [Liu, 2008].

For checking consistency of compositions, if SPR_1 contains a branching, it does not suffice for SPR_2 (which must be a component refinement) to be consistent, because some component of SPR_1 outside the domain of SPR_2 might be inconsistent.

Theorem 11.5.1 (Soundness) *If $\vdash SPR ::_c \square$, the calculi for checking consistency of structured specifications and conservativity of extensions are sound and $\vdash \text{cons}(SPR)$, then SPR has a model.*

Proof. Induction on the structure of SPR .

Case $SPR = USP$ follows from the soundness of the calculi for consistency and conservativity.

Case $SPR = USP$ refined via σ to SPR' . By definition, $\vdash \text{cons}(SPR)$ holds if $\vdash \text{cons}(SPR')$ holds. By the induction hypothesis, the model class \mathcal{R}' of SPR' is non-empty. Since $\vdash SPR ::_c \square$, it follows that $USP \rightsquigarrow^\sigma USP'$, where $\vdash SPR' ::_c (USP', BSP)$. This means that $U|_\sigma \in \text{Unit}(USP)$ for any $(U, BM) \in \mathcal{R}'$. It follows that the model class $\mathcal{R} = \{(U|_\sigma, BM) \mid (U, BM) \in \mathcal{R}'\}$ of SPR is non-empty.

Case $SPR = \{UN_i \text{ to } SPR_i\}_{i \in \mathcal{J}}$. By the hypothesis we get that $\vdash \text{cons}(SPR_i)$ and $\vdash SPR_i ::_c \square$, for any $i \in \mathcal{J}$. By the induction hypothesis, for any $i \in \mathcal{J}$, the model class \mathcal{R}_i of SPR_i is non-empty. We can therefore take a model from each such class and combine them like in the rule for component refinements in Fig. 5.3 to obtain a model of SPR .

Case $SPR = SPR_1 \text{ then } SPR_2$ has two sub-cases.

Let us first assume that SPR_1 contains branchings. By hypothesis we get that $\vdash \text{cons}(SPR_i)$ and $\vdash SPR_i ::_c \square$, for any $i \in \{1, 2\}$. By induction hypothesis, we get that the model classes \mathcal{R}_1 and \mathcal{R}_2 of SPR_1 and SPR_2 respectively are non-empty. Since $\vdash SPR ::_c \square$ and SPR_1 and SPR_2 are consistent, using the soundness of the proof calculus we have that $\mathcal{R}_1; \mathcal{R}_2$ is defined and $\vdash SPR \Rightarrow \mathcal{R}_1; \mathcal{R}_2$. Since the composition $\mathcal{R}_1; \mathcal{R}_2$ is defined, there must be a model $R'_1 = (U, BE) \in \mathcal{R}_1$ such that R_2 matches the branching environment BE of R'_1 and thus $(U, BE[R_2])$ is defined and gives us a model of $\mathcal{R}_1; \mathcal{R}_2$.

Let us then consider the case when SPR_1 does not contain branchings. By hypothesis we get that $\vdash \text{cons}(SPR_2)$ and $\vdash SPR_i ::_c \square$, for any $i \in \{1, 2\}$. By induction we get that the model class \mathcal{R}_2 of SPR_2 is non-empty. Notice that since SPR_1 does not contain branchings, it must be the case that $\vdash SPR_1 \triangleright (U\Sigma, U\Sigma')$ and $\vdash SPR_2 \triangleright (U\Sigma', B\Sigma)$. Then, since the constructor κ associated with SPR_1 is total, we can define $\mathcal{R} = \{(\kappa(U), BM) \mid (U, BM) \in \mathcal{R}_2\}$ and, since \mathcal{R}_2 is non-empty, so is \mathcal{R} .

Case $SPR = \text{arch spec } ASP$. By induction hypothesis, SPR_i has a model, for all $UN_i : SPR_i$ in ASP . This gives us a unit environment for ASP by projecting each of the models of SPR_i to the first component and we can combine the units in the way described by the result unit of ASP to get a model of ASP because ASP is statically correct. □

Completeness holds again only if the specification of each unit term is not approximating, but exactly capturing the model class of the unit term.

Theorem 11.5.2 (Completeness) *If no generic unit is applied more than once, the calculi for checking consistency of structured specifications and conservativity of extensions are complete, $\vdash SPR ::_c \square$ and SPR has a model, then $\vdash \text{cons}(SPR)$.*

Proof. Structural induction on SPR .

Case $SPR = USP$ follows from the completeness of the calculus for structured specifications and conservativity of extensions of structured specifications.

Case $SPR = USP$ refined via σ to SPR' . Let \mathcal{R} be the model class of SPR . With the corresponding model semantics rule, $\mathcal{R} = \{(U|_\sigma, BM) \mid (U, BM) \in \mathcal{R}'\}$

where \mathcal{R}' is the model class of SPR' . Since $\vdash SPR ::_c \square$ we also get $\vdash SPR' ::_c \square$ and non-emptiness of \mathcal{R} implies non-emptiness of \mathcal{R}' . We can apply the inductive hypothesis to get $\vdash cons(SPR')$, which implies $\vdash cons(SPR)$.

Case $SPR = \{UN_i \text{ to } SPR_i\}_{i \in \mathcal{J}}$. If SPR has a model, then we can define models of SPR_i by projecting the model of SPR to the i -th component. Since correctness of SPR implies correctness of SPR_i , by the induction hypothesis we obtain $\vdash cons(SPR_i)$. With the rule for component refinements we get that $\vdash cons(SPR)$.

Case $SPR = SPR_1 \text{ then } SPR_2$. If SPR has a non-empty model class \mathcal{R} , let $R \in \mathcal{R}$. With the corresponding model semantics rule there must be models $R_1 \in \mathcal{R}_1$ and $R_2 \in \mathcal{R}_2$ such that $R = R_1; R_2$. By induction hypothesis we get $\vdash cons(SPR_1)$ and $\vdash cons(SPR_2)$ and with the calculus rule for compositions of refinements we get $\vdash cons(SPR)$. Note that if SPR_1 has no branching, it suffices to use $\vdash cons(SPR_2)$.

Case $SPR = \text{arch spec } ASP$. If ASP has a model class \mathcal{AM} , then by the model semantics rule we know that for any unit declaration $UN_i : SPR_i$ of ASP , SPR_i has a non-empty model class \mathcal{R}_i and moreover $\mathcal{AM} = \{(U, \pi_2(BM)) \mid BM(UN_i) \in \mathcal{R}_i, U \text{ combines the units } \pi_1(BM) \text{ according to the result unit of } ASP\}$. We can apply the induction hypothesis to get $\vdash cons(SPR_i)$ and with the rule for architectural specifications we get $\vdash cons(ASP)$.

□

Refinement trees prove useful for making consistency checks with HETS. Checking consistency has been added as a context menu option for nodes in refinement trees: in the case of architectural specifications, the branching points in refinement trees provide the appropriate representation. Selecting 'Check consistency' leads to introducing consistency obligations in the development graph of the specification: nodes corresponding to non-generic units carry consistency proof obligations (marked with color yellow in HETS), while morphism corresponding to theory extensions of generic units carry conservativity obligations (marked with the label 'Cons?' on the edge). If the node is a branching point, consistency is checked recursively for all components. If an edge in the refinement tree is a refinement link, it suffices to check consistency of the target of the edge. HETS can be further employed for discarding these obligations, by making use of the model finders it interfaces, e.g. Isabelle-refute [Weber, 2005], Darwin [Baumgartner et al., 2004], or SPASS [Weidenbach et al., 2002], and also the conservativity checker of [Liu, 2008].

11.6 Conclusion and Future Work

We stress that both the language for refinements, as well as its semantics, the notion of refinement tree and the (sound and complete) proof calculus for refinements are given in an *institution-independent* way; that is, it applies to any logic that satisfies very mild conditions. In particular, we could use in principle the Grothendieck institution (Sec. 3.3), thus obtaining *heterogeneous refinement*. While this can be done with no problem for the simple refinements, using heterogeneous signatures

morphisms, as we will already do in the next chapter, an open problem remains making the architectural specifications heterogeneous. Recall that in Chapter 8 we constructed an algorithm for computing approximations of heterogeneous colimits that could be employed in verification of such architectural specifications. The question is then how to discharge statically the amalgamability conditions in the semantics of architectural specifications [Mosses, 2004] for a heterogeneous diagram, assuming that this can be done for the individual logics involved.

We also support refinements trees practically: we have implemented them in the Heterogeneous Tool Set HETS, such that browsing through and inspection of complex formal developments becomes possible. An implementation of the proof calculus is currently in progress; the refinement part is already implemented. Note that the proof calculus for architectural specifications of [Mosses, 2004] was given for a restricted version of the language; it can be extended to the whole language in a way substantially simplified by the transformation of units with imports into generic units. We also have introduced and implemented a sound and complete calculus for consistency of refinements and architectural specification, which already has been applied for proving the consistency of the upper ontology Dolce in a modular way.

One problem with the approach described so far is that the constructors provided by specification morphisms and architectural specifications in CASL do not suffice for implementing specifications. In a sense, these constructors only provide means to *combine* or *modify* existing program units—but there is no way to build program units *from scratch*. That is, CASL lacks a notion of *program*. A discussion on adding programs in CASL can be found in [Mossakowski et al., 2005].

Future work includes extending the language to support *behavioural refinement*, corresponding to *abstractor implementations* in [Sannella and Tarlecki, 1988a]. Often, a refined specification does not satisfy the initial requirements literally, but only up to some sort of behavioural equivalence: for example, if stacks are implemented as arrays-with-pointer, then two arrays-with-pointer differing only in their “junk” entries (that is, those that are “above” the pointer) exhibit the same behaviour in terms of the stack operations, and hence correspond to the same abstract stack. This can be taken into account by re-interpreting unit specifications to include models that are behaviourally equivalent to literal models, see [Bidoit et al., 2002b, 2008]; then specification refinements as considered here become behavioural. The next chapter provides a first step in this direction.

Another useful addition would be amalgamability checks for other logics than CASL in the Hets’ logic graph, making thus possible to have architectural specifications in those logics.

Acknowledgement. This chapter extends [Codescu and Mossakowski, 2011] with more detailed proofs and with the discussion on completeness of the proof calculus for refinements in Sec. 11.3. The idea of defining the specification of a unit term (Dfn. 11.1.6) and the proof of Thm. 11.1.12 are due to T. Mossakowski. My contribution includes the definition of refinement trees in Sec. 11.4, the proof calculi for refinement in Sec. 11.2 and the consistency calculus in Sec. 11.5, as

well as the implementation of the static analysis and calculi for refinement and consistency in HETS. The presentation has greatly benefited from comments of A. Tarlecki.

VSE Refinement in HETS

Contents

12.1 Presentation of VSE	196
12.2 Institution of Dynamic Logic	198
12.2.1 Signatures	198
12.2.2 Sentences	199
12.2.3 Models	201
12.2.4 Satisfaction of Dynamic Logic Formulas	201
12.2.5 Satisfaction of Procedure Definitions	203
12.2.6 Satisfaction of restricted sort generation constraints	203
12.2.7 Satisfaction condition	204
12.3 VSE Refinement as an Institution Comorphism	205
12.3.1 The Refinement Comorphism	207
12.3.2 Structuring in Context of Refinement	211
12.4 Example: Implementing natural numbers by binary words	213
12.5 Conclusions and future work	219

Axiomatic specification of data and programs provide the means for developing formal models of software at a conceptual level, while dynamic logics and Hoare-style logics can express correctness criteria that stay closer to the actual programs. For a formal development or verification of software, typically both levels are needed, since using axiomatic modeling of the concepts alone misses the formal link to real programs, while using a Hoare-style or dynamic logic alone only allows for little formal conceptual modeling. Therefore, we integrate HETS with the specification environment Verification Support Environment (VSE) [Autexier et al., 2000], developed at DFKI Saarbrücken, which provides an industrial-strength methodology for specification and verification of imperative programs.

We want to combine the best of both worlds by establishing a connection between the VSE prover and the HETS proof management. For VSE, this brings additionally flexibility: VSE specifications can now be verified not only with the VSE prover, but also with provers like the first-order prover SPASS [Weidenbach et al., 2002] and the higher-order prover Isabelle [Nipkow et al., 2002] which are interfaced with HETS. On the other hand, HETS benefits from VSE's industrial experience, including a practical relation between specification and programming languages together with the necessary proof support. Being interactive, the VSE prover

offers enough flexibility to tackle even challenging proof obligations, while a set of strong heuristics based on symbolic execution provide automation to keep the proof effort still small. VSE provides also a code generation mechanism to imperative programming languages like Ada or C.

The benefit of plugging VSE into HETS is that for both verification and refinement, we can use the general proof management mechanisms of the HETS motherboard, instead of the specialized refinement tools hard-wired into VSE. Moreover, the HETS motherboard already has plugged in a number of expansion cards (e.g., the theorem provers Isabelle, SPASS and more, as well as model finders) that can be used for VSE as well. The challenge is that typically, analysis and proof tools that shall be plugged into the HETS motherboard are not compatible with HETS expansion slots. Often, this is a matter of writing a suitable wrapper that encapsulates the tool in an expansion card that is compatible to the HETS motherboard. However, sometimes also the specification of the expansion slot has to be enhanced. Of course, such enhancements should only be done for very good reasons — otherwise, one will end up with slots containing hundreds of special pins. Since VSE provides a special notion of refinement, one is tempted to enhance the specification of the expansion slot in this case. However, we will see that we can do without such an enhancement.

Related work includes ad-hoc integration of (tools for) formal methods, see e.g. the integrated formal methods conference series [Leuschel and Wehrheim, 2009], and integrations of decision procedures, model checkers and automated theorem provers into interactive theorem provers [Dennis et al., 2003, Meng et al., 2006]. However, these approaches are not as flexible as the HETS motherboard/expansion card mechanism. In many approaches, the interfaces for these integrations are ad-hoc and not re-used in many different contexts. Moreover, we will see in Sec. 12.3 below that the use of *logic translations as first class citizens* in the expansion card mechanism is crucial for integrating VSE and HETS in a modular way. This clearly is a novel feature of our approach.

12.1 Presentation of VSE

The Verification Support Environment (VSE) is a tool that supports the formal development of complex large scale software systems from abstract high level specifications down to the code level. It provides both an administration system to manage structured formal specifications and a deductive component to maintain correctness on the various abstraction levels (see Fig. 12.1). Taken together, these components guarantee the overall correctness of the complete development. The structured approach allows the developer to combine specifications in an algebraic functional style with state-based formal descriptions of concurrent systems.

VSE has been developed in two phases on behalf the German Bundesamt für Sicherheit in der Informationstechnik (BSI) to satisfy the needs in software developments according to the upcoming standards ITSEC and Common Criteria. Since

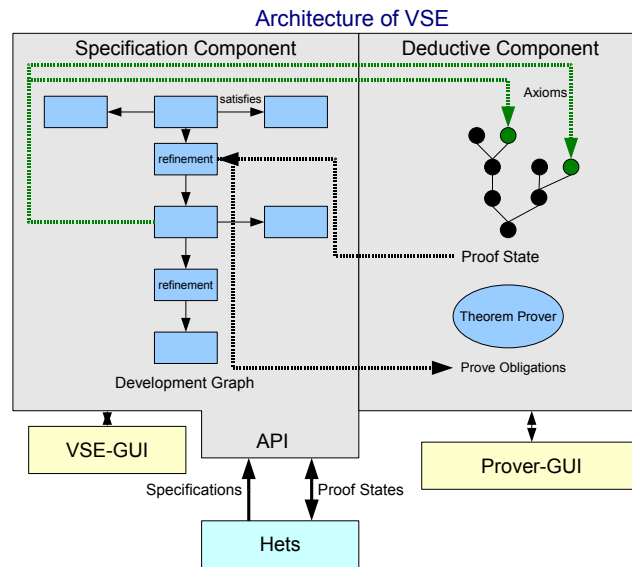


Figure 12.1: Architecture of VSE

then, VSE has been successfully applied in several industrial and research projects, many of them being related to software evaluation [Hutter et al., 2000, Autexier et al., 2000, Langenstein et al., 2000, Cheikhrouhou et al., 2006]. The models¹ developed with VSE comprise among others the control system of a heavy robot facility, the control system of a storm surge barrier, a formal security policy model conforming to the German signature law and protocols for chip card-based biometric identification.

VSE supports a development process that starts with a modular formal description of the *system model* possibly together with separate *requirements* or *security objectives*. Logically the requirements have to be derivable from the system model. Therefore, the requirements lead to proof obligations that must be discharged by using the integrated deductive component of VSE.

In a *refinement process* the abstract system model can be related to more concrete models. This is in correspondence with a software development that starts from a high-level design and then descends to the lower software layers such that in a sense higher layers are implemented based on lower layers. Each such step can be reflected by a *refinement step* in VSE. These steps involve programming notions in the form of abstract implementations, that can later be exploited to generate executable code. Each refinement step gives rise to proof obligations showing the correctness of the implementations. Refinements also can be used to prove consistency of specifications, because they describe a way how to construct a model. This plays a major role for the formal specifications required for Common Criteria,

¹This use of the term “model” is in the sense of modeling, while the institutional use is in the sense of logic and model theory, see Sect. 12.2.3.

which only need to cover higher abstraction levels.

In addition to the *vertical* structure given by refinement steps, VSE also allows the specification to be structured *horizontally* to organize the specifications on one abstraction level. Each single (sub)specification can be refined vertically or further decomposed horizontally, such that the complete development is represented by a *development graph*. The deductive component is aware of this structure. This is an important aspect for the interactive proof approach, as the structure helps the user to prove lemmas or proof obligations that require properties from various parts of the specification.

12.2 Institution of Dynamic Logic

VSE provides an interactive prover, which supports a Gentzen-style natural deduction calculus for dynamic logic. This logic is an extension of first-order logic with two additional kinds of formulas that allow for reasoning about programs. One of them is the box formula $[\alpha]e$, where α is a program written in an imperative language, and e is a dynamic logic formula. The meaning of $[\alpha]e$ can be roughly put as “After every terminating execution of α , e holds.” The other new kind of formulas is the diamond formula $\langle\alpha\rangle e$, which is the dual counter part of a box formula. The meaning of $\langle\alpha\rangle e$ can be described as “After some terminating execution of α , e holds”.

We will now describe the formalization of this dynamic logic as an institution, denoted $CDyn^=$, in detail, because this has not been done in the literature so far.

12.2.1 Signatures

The starting point for dynamic logic signatures are the signatures of first-order logic with equality ($FOL^=$) that have the form $\Sigma_{FOL^=} = (S, F, P)$ consisting of a set S of sorts, a family F of function symbols and a family P of predicate symbols. Because we need to name procedures, we add an $S^* \times S^*$ -sorted family $PR = (PR_{v,w})_{v,w \in S^*}$ of procedure symbols, leading to signatures of the form $\Sigma = (S, F, P, PR)$. We have two separate lists v and w of the argument sorts of the procedure symbols in $PR_{v,w}$, in order to distinguish the sorts of the input parameters (v) from those of the output parameters (w)². When the string of output parameters consists of just one sort s , we can mark some of the procedures of $PR_{v,s}$ as *functional procedures* and we denote this subset as $FP_{v,s}$.

A signature morphism between two signatures maps sorts, operation symbols, predicate symbols and procedure symbols in a way such that argument and result sorts are preserved. Also, signature morphisms are required to map functional

²In VSE syntax, the input parameters of the procedures are preceded by **IN** and the output parameters, by **OUT**. In the case of functional procedures, since all parameters except the last are input parameters, these annotations are not present.

procedures to functional procedures and the mapping between procedure symbols must be injective.

Moreover, it is assumed that all signatures have a sort *Boolean* together with two constants *true* and *false* on it and this subsignature is preserved by signature morphisms.

12.2.2 Sentences

Let $\Sigma = (S, F, P, PR)$ be a dynamic logic signature with $PR = (PR_{v,w})_{v,w \in S^*}$. The variables will be taken from an arbitrary but fixed countably infinite set \mathbb{X} which is required to be closed under disjoint unions.

First we define the syntax of the programs that may appear in dynamic logic formulas. The programs contain Σ -terms, which are predicate logical terms of $(S, (F_{v,s} \cup FP_{v,s})_{v \in S^*, s \in S}, P)$, i.e. in addition to variables and function symbols we allow symbols of *functional* procedures to occur in these terms. The set \mathbb{P}_Σ of Σ -programs is the smallest set containing:

- **abort**
- **skip**
- $x := \tau$
- **declare** $x : s = \tau$
- **declare** $x : s = ?$
- $\alpha; \beta$
- **if** ε **then** α **else** β **fi**
- **while** ε **do** α **od**
- $p(x_1, x_2, \dots, x_n; y_1, y_2, \dots, y_m)$,

where $x, x_1, x_2, \dots, x_n \in \mathbb{X}$ are variables, $y_1, y_2, \dots, y_m \in \mathbb{X}$ are pairwise different variables, τ a Σ -term of sort s , ε a boolean Σ -formula (i.e. a Σ -formula without quantifiers, boxes and diamonds)³, $\alpha, \beta \in \mathbb{P}_\Sigma$, p a procedure symbol, such that the sorts of $x_1, \dots, x_n, y_1, \dots, y_m$ match the argument and result sorts of p . Moreover, in the case of functional procedures, programs also contain **return** τ , where τ is a term of the result sort of the functional procedure.

These kinds of program statements can be explained informally as follows: **abort** is a program that never terminates. **skip** is a program that does nothing. $x := \tau$ is the assignment. **declare** $x : s = \tau$ is the deterministic form of a variable declaration which sets x to the value of τ . Its nondeterministic form

³This restriction is motivated by the straightforward translation of such formulas into program expressions.

declare $x : s = ?$ sets x to an arbitrary value.⁴ The nondeterministic declaration can not be used for functional procedures. $\alpha; \beta$ is the composition of the programs α and β , such that α is executed before β . The *conditional* **if** ε **then** α **else** β **fi** means that α is executed if ε holds, otherwise β is computed. The *loop* **while** ε **do** α **od** checks the condition ε , in case of validity executes α and repeats the loop. Finally, $p(x_1, x_2, \dots, x_n; y_1, y_2, \dots, y_m)$ calls the procedure p with input parameters x_1, x_2, \dots, x_n and output parameters y_1, y_2, \dots, y_m .

There are three kinds of sentences that may occur in a Σ -dynamic logic specification.

1. The set of dynamic logic Σ -formulas is the smallest set containing
 - *True* and *False*
 - the (S, F, P) -first-order formulas e ;
 - for any dynamic logic Σ -formulas e, e_1, e_2 , any variable x , and any sort $s \in S$ and any Σ -program α the formulas $[\alpha]e$, $\langle \alpha \rangle e$ and $\neg e$, $e_1 \wedge e_2$ and $\forall x : s.e$.
2. Procedure definitions are expressions of the form:

```

defprocs
  procedure  $pr_1(x_1^1, \dots, x_{n_1}^1, y_1^1, \dots, y_{m_1}^1)\alpha_1$ 
  ...
  procedure  $pr_k(x_1^k, \dots, x_{n_k}^k, y_1^k, \dots, y_{m_k}^k)\alpha_k$ 
defprocsend

```

where $pr_i \in PR_{v_i, w_i}$ for some $v_i, w_i \in S^*$, $x_1^i, \dots, x_{n_i}^i, y_1^i, \dots, y_{m_i}^i$ are variables of the corresponding sorts in v_i, w_i , and $\alpha_i \in \mathbb{P}_\Sigma$ is a Σ -program with free variables from $\{x_1^i, \dots, x_{n_i}^i, y_1^i, \dots, y_{m_i}^i\}$. In VSE the functional procedures are written using **function** rather than **procedure** and without a formal output parameter; to simplify notation, we will ignore this in this section.

3. *Restricted sort generation constraints* express that a set of values defined by restriction procedure can be generated by the given set of procedures, the *constructors*. Syntactically a restricted sort generation constraints takes the form

```

generated types
 $s_1 ::= p_1^1(\dots) | p_2^1(\dots) | \dots | p_n^1(\dots)$  restricted by  $r^1$  ,
...
 $s_k ::= p_1^k(\dots) | p_2^k(\dots) | \dots | p_n^k(\dots)$  restricted by  $r^k$  ,

```

where s_i are sort symbols, p_1^i, \dots, p_n^i are functional procedure symbols, the dots in $p_j^i(\dots)$ etc. have to be replaced by a list of the argument sorts, and r^i is a procedure symbol taking one argument of sort s_i .

⁴In VSE one can also declare more than one variable in a **declare** list; for simplicity we restrict to the case of a single variable.

In order to make the satisfaction condition hold, we formally define a sort generation constraint over a signature Σ as a tuple (S', F', PR', ϑ) , where $\vartheta : \Sigma_0 = (S_0, F_0, P_0, PR_0) \rightarrow \Sigma$, $S' \subseteq S_0$, $F' \subseteq F_0$ and $PR' \subseteq PR_0$ such that in PR' there is a restriction procedure r_i for any sort s_i in S' .

For any signature morphism $\sigma : \Sigma \rightarrow \Sigma'$, the translation of Σ -sentences along σ is done by translating each symbol according to the sort, operation symbol, predicate symbol and procedure symbol mappings respectively. In the case of quantified sentences, $\forall X.e$ gets mapped to $\forall X'.\sigma(e)$, where for any sort s' of Σ' , $X'_{s'}$ is defined as the disjoint union of X_s for all s such that $\sigma(s) = s'$. The translation of a sort generation constraint (S', F', PR', ϑ) over Σ along σ is defined as $(S', F', PR', \vartheta; \sigma)$.

12.2.3 Models

Let $\Sigma = (S, F, P, PR)$ be a dynamic logic signature with $F = (F_{w,s})_{w \in S^*, s \in S}$, $P = (P_w)_{w \in S^*}$, $PR = (PR_{v,w})_{v,w \in S^*}$. A (dynamic logic) Σ -model M maps each sort symbol $s \in S$ to a carrier set M_s , each function symbol $f \in F_{w,s}$ to a total function $M_f : M_w \rightarrow M_s$, each predicate symbol $p \in P_w$ to a relation $M_p \subseteq M_w$ and each procedure symbol $pr \in PR_{v,w}$ to a relation $M_{pr} \subseteq M_v \times M_w$, where $M_{(s_1, \dots, s_n)}$ denotes $M_{s_1} \times M_{s_2} \times \dots \times M_{s_n}$ for $(s_1, s_2, \dots, s_n) \in S^*$. Functional procedures are required to be interpreted as total functions over their domain. Thus, such a model can be viewed as a $CFOL^\equiv$ structure extended with the interpretation of procedure symbols.

For any signature morphism $\sigma : \Sigma \rightarrow \Sigma'$, the reduct $M'|_\sigma$ of a Σ' -model M' interprets x as the interpretation of $\sigma(x)$ in the original model, where x can be either a sort, a function symbol, a predicate symbol or a procedure symbol.

12.2.4 Satisfaction of Dynamic Logic Formulas

Semantics is defined in a Kripke-like manner. For a given signature Σ and a Σ -model M the (program) states are variable valuations, i.e. partial functions taking sorted variables $x : s$ to values of M_s where s is a sort of Σ and $x \in \mathbb{X}$. We assume that for each sort there is a designated output variable for storing the return values of functional procedures; let us denote that variable o .

First, we need to define the interpretation of a term τ in a model M and a state q . Because the states are partial, the interpretation can be undefined. The interpretation of terms is then done as usual inductively on the structure of terms:

- if τ is a variable $x : s$, then $\tau^{M,q} := q(x : s)$ (i.e. the value of the variable $x : s$ in state q when defined and undefined otherwise);
- if $\tau = f(\tau_1, \dots, \tau_n)$ and $f \in F_{w,s}$ or $f \in FP_{w,s}$, then $\tau^{M,q} := M_f(\tau_1^{M,q}, \dots, \tau_n^{M,q})$, and undefinedness of any of the $\tau_i^{M,q}$ is propagated;

The semantics of a program α with respect to a model M is a predicate $\llbracket \alpha \rrbracket^M$ on two program states. $q \llbracket \alpha \rrbracket^M r$ can be read as: If α is started in state q it may terminate after having changed the state to r .

- $q \llbracket \text{skip} \rrbracket^M q$
- not $q \llbracket \text{abort} \rrbracket^M r$
- $q \llbracket x := \tau \rrbracket^M r \Leftrightarrow r = q[x : s \leftarrow \tau^{M,q}]$ and $\tau^{M,q}$ is defined, where $s = \text{sort}(\tau)$
- $q \llbracket x := \tau \rrbracket^M r$ does not hold for any r if $\tau^{M,q}$ is not defined
- $q \llbracket \alpha; \beta \rrbracket^M r \Leftrightarrow$ for some state $s : q \llbracket \alpha \rrbracket^M s$ and $s \llbracket \beta \rrbracket^M r$
- $q \llbracket \text{declare } x : s = \tau \rrbracket^M r \Leftrightarrow q \llbracket x := \tau \rrbracket^M r$
- $q \llbracket \text{declare } x : s = ? \rrbracket^M r \Leftrightarrow$ for some $a \in s^M : r = q[x \leftarrow a]$
- $q \llbracket \text{if } \varepsilon \text{ then } \alpha \text{ else } \beta \text{ fi} \rrbracket^M r \Leftrightarrow (q \models \varepsilon \text{ and } q \llbracket \alpha \rrbracket^M r) \text{ or } (q \models \neg \varepsilon \text{ and } q \llbracket \beta \rrbracket^M r)$
- $q \llbracket \text{while } \varepsilon \text{ do } \alpha \text{ od} \rrbracket^M r \Leftrightarrow q(\llbracket \text{if } \varepsilon \text{ then } \alpha \text{ else skip fi} \rrbracket^M)^* r$ and $r \models \neg \varepsilon$
- $q \llbracket \text{pr}(x_1, \dots, x_n; y_1, \dots, y_m) \rrbracket^M r \Leftrightarrow \text{pr}_M(q(x_1), \dots, q(x_n); r(y_1), \dots, r(y_m))$
- $q \llbracket \text{return } \tau \rrbracket^M r \Leftrightarrow r = q[o : s \leftarrow \tau]$, where o is the output variable of sort $s = \text{sort}(\tau)$.

where for any program α , $(\llbracket \alpha \rrbracket^M)^*$ is the reflexive transitive closure of the relation $\llbracket \alpha \rrbracket^M$, and the state $q[x \leftarrow a]$ is defined as $q[x \leftarrow a](y) = \begin{cases} q(y) & \text{if } y \neq x \\ a & \text{if } y = x \end{cases}$, τ is a Σ -term without predicate symbols and $\tau^{M,q}$ is the evaluation of the term τ with respect to the model M and state q .

We define satisfaction on a model M and a program state r as follows:

- $M, r \models \text{True}$ and $M, r \not\models \text{False}$
- $M, r \models p(\tau_1, \dots, \tau_n) \Leftrightarrow$ for all $i = 1, \dots, n$, $\tau_i^{M,r}$ is defined and $M_p(\tau_1^{M,r}, \dots, \tau_n^{M,r})$
- $M, r \models \tau_1 = \tau_2 \Leftrightarrow \tau_1^{M,r} = \tau_2^{M,r}$ and interpretation of both terms in the state r is defined
- $M, r \models \neg e \Leftrightarrow M, r \not\models e$
- $M, r \models e \wedge e' \Leftrightarrow M, r \models e$ and $M, r \models e'$
- $M, r \models e \vee e' \Leftrightarrow M, r \models e$ or $M, r \models e'$
- $M, r \models \forall x : s.e \Leftrightarrow$ for all $a \in M_s : M, r[x : s \leftarrow a] \models e$
- $M, r \models [\alpha]e \Leftrightarrow$ for all program states q with $r \llbracket \alpha \rrbracket^M q : M, q \models e$

The formula $\langle \alpha \rangle e$ is to be read as an abbreviation for $\neg[\alpha]\neg e$. Finally a formula e holds on a model M ($M \models e$), if for all program states r it holds on M and r ($M, r \models e$).

12.2.5 Satisfaction of Procedure Definitions

The procedures in our model will not have any side effects (except for modifying the output parameters).

Unwinding a procedure call by replacing it by the body of the procedure and substituting the formal parameter variables by the actual parameters should not change the result of a program. Therefore, for a signature Σ , a Σ -model M is a model of a procedure declaration without recursion

```

defprocs
  procedure  $pr_1(x_1^1, \dots, x_{n_1}^1, y_1^1, \dots, y_{m_1}^1)\alpha_1$ 
  ...
  procedure  $pr_k(x_1^k, \dots, x_{n_k}^k, y_1^k, \dots, y_{m_k}^k)\alpha_k$ 
defprocsend

```

if

$$M \models \forall x_1^i, \dots, x_{n_i}^i, r_1^i, \dots, r_{m_i}^i : \\ (\langle pr(x_1^i, \dots, x_{n_i}^i; y_1^i, \dots, y_{m_i}^i) \rangle y_1^i = r_1^i \wedge \dots \wedge y_{m_i}^i = r_{m_i}^i) \Leftrightarrow \langle \alpha \rangle y_1^i = r_1^i \wedge \dots \wedge y_{m_i}^i = r_{m_i}^i$$

holds for any $i = 1 \dots k$. Abbreviating the procedure declaration as Π , we then write $M \models \Pi$.

In the presence of recursion this is not sufficient to make the procedure definitions non-ambiguous and adequate to conventional semantics of programming languages. Therefore, from several models complying with the definitions the minimal model with respect to some order will be chosen. The order compares the interpretations of the procedures symbols, such that the order relation $M_1 \leq_{\Pi} M_2$ holds for two models M_1 and M_2 for the same signature $\Sigma = (S, F, P, PR)$ iff $pr_i^{M_1} \subseteq pr_i^{M_2}$ for all procedure symbols pr , and the interpretations of sort, function, predicate symbols and procedure symbols which are not part of Π are identical. Moreover, we say that a model M_2 is a Π -variant of M_1 , written $M_1 \equiv^{\Pi} M_2$, if M_1 and M_2 agree on the interpretations of all symbols except possibly the procedure symbols in Π . Then we define that the satisfaction of a procedure declaration Π by M as follows:

$$M \models \Pi \quad \text{iff} \quad M \models \Pi \text{ and for all } \Pi\text{-variants } M' \text{ of } M, M' \models \Pi \text{ implies } M \leq_{\Pi} M'.$$

12.2.6 Satisfaction of restricted sort generation constraints

A restricted sort generation constraint (S', F', PR', ϑ) written as

generated types $s_i ::= p_1^i(\dots) | p_2^i(\dots) | \dots | p_n^i(\dots)$ **restricted by** r^i ,

is said to hold in a model M , if the subset of the carrier $(M|_{\vartheta})_{s_i}$ on which the restriction procedure r^i terminates is generated by the functional procedures $p_1^i, p_2^i, \dots, p_n^i$ (called *constructor procedures*). In more detail: for each element a of $(M|_{\vartheta})_{s_i}$ such that $M|_{\vartheta}, q \models \langle r^i(x) \rangle true$ when q is a state such that $q(x) = a$, there must be a term t built with constructor procedures only and having no free variables of sorts s_i and a state v such that v is defined only for variables of sorts distinct from the s_i and that $t^{M|_{\vartheta}, v} = a$ holds.

12.2.7 Satisfaction condition

Proposition 12.2.1 *Let $\Sigma = (S, F, P, PR)$ and $\Sigma' = (S', F', P', PR')$ be two dynamic logic signatures, $\sigma : \Sigma \rightarrow \Sigma'$ a signature morphism, M' a Σ' -model and e a Σ -sentence. Then:*

$$M' \models \sigma(e) \iff M'|_{\sigma} \models e$$

Proof. We prove the satisfaction condition by case distinction on e .

1. e is a dynamic logic formula.

In this case, we prove by induction over e that $M', t' \models \sigma(e) \iff M'|_{\sigma}, t'|_{\sigma} \models e$, where for any state t' for Σ' and M' , we define the state $t'|_{\sigma}$ for Σ and $M'|_{\sigma}$ by taking $t'|_{\sigma}(x : s) = t'(x : \sigma(s))$ for each sort s of Σ . The proof is pretty much routine, the interesting case being when e is of form $[\alpha]e'$. Let us denote $M = M'|_{\sigma}$.

We begin with a lemma:

Lemma 12.2.2 *For any state t' for Σ' and M' , any state q for Σ and M and any Σ -program α , $t'|_{\sigma} \llbracket \alpha \rrbracket^M q$ iff there is a state q' such that $t' \llbracket \sigma(\alpha) \rrbracket^{M'} q'$ and $q'|_{\sigma} = q$.*

which can be proven by induction on α , by making use of the fact that the variables used in α are bijectively renamed in the states q and q' .

Let t' be a state for Σ' and M' such that $M', t' \models \sigma([\alpha]e')$. By definition this means that for any state p' such that $t' \llbracket \sigma(\alpha) \rrbracket^{M'} p'$, $M', p' \models \sigma(e')$. Let q be a state such that $t'|_{\sigma} \llbracket \alpha \rrbracket^M q$. We need to prove that $M, q \models e'$. Using Lemma 12.2.2, there is a state q' such that $t' \llbracket \sigma(\alpha) \rrbracket^{M'} q'$ and $q'|_{\sigma} = q$. By the hypothesis we get that $M', q' \models \sigma(e')$. By the inductive hypothesis for e' , we obtain $M, q'|_{\sigma} \models e'$. Since $q'|_{\sigma} = q$, this means $M, q \models e'$. Since q was arbitrary such that $t'|_{\sigma} \llbracket \alpha \rrbracket^M q$, we obtain $M, t'|_{\sigma} \models [\alpha]e'$.

For the reverse implication, let t' be a state for Σ' and M' such that $M, t'|_{\sigma} \models [\alpha]e'$. By definition this means that for any state p such that $t'|_{\sigma} \llbracket \alpha \rrbracket^M p$, $M, p \models e'$. Let q' be a state such that $t' \llbracket \sigma(\alpha) \rrbracket^{M'} q'$. By Lemma 12.2.2 we

get $t' |_{\sigma} \llbracket \alpha \rrbracket^M q' |_{\sigma}$. By the hypothesis we get that $M, q' |_{\sigma} \models e'$. By the inductive hypothesis for e' we get that $M', q' \models \sigma(e')$ and since q' was arbitrary, by definition $M', t' \models [\sigma(\alpha)]\sigma(e')$.

2. e is a procedure definition.

We first prove the following lemma:

Lemma 12.2.3 *Let $\sigma : \Sigma \rightarrow \Sigma'$, let Π be a procedure definition in Σ and let N be a Σ' -model, and let $M = N |_{\sigma}$. Then*

$$\begin{aligned} N \leq_{\sigma(\Pi)} N' \text{ for any } \sigma(\Pi)\text{-variant } N' \text{ of } N \text{ such that } N' \models \sigma(\Pi) \\ \text{iff } M \leq_{\Pi} M' \text{ for any } \Pi\text{-variant } M' \text{ of } M \text{ such that } M' \models \Pi. \end{aligned}$$

Proof: For the left to right implication, assume for a contradiction that M is not minimal among its Π -variants satisfying Π . Then there exists a Π -variant of M , M^0 , satisfying Π such that $M \not\leq_{\Pi} M^0$. We define a σ -expansion N^0 of M^0 by interpreting all symbols outside Π as in the model N and taking $N_{\sigma(\pi)}^0 = M_{\pi}^0$ for any procedure π defined in Π . The well-definedness is ensured by $M^0 \models \Pi$ and by the injectivity of σ on procedure symbols and moreover, by the satisfaction condition we get that $N^0 \models \sigma(\Pi)$. Since $N \leq_{\sigma(\Pi)} N^0$, we get a contradiction with the minimality of N .

For the right to left implication, assume for a contradiction that N is not minimal. Then there exists a $\sigma(\Pi)$ -variant of N , denoted N^0 such that $N \not\leq_{\sigma(\Pi)} N^0$. Let $M^0 := N^0 |_{\sigma}$. By the satisfaction condition we get $M^0 \models \Pi$. By definition of reduct it follows that $M_{\pi}^0 = N_{\sigma(\pi)}^0$ and since $M_{\pi} = N_{\sigma(\pi)}$ we get $M \not\leq_{\Pi} M^0$ which contradicts the minimality of M .

□

The satisfaction condition follows from the definition of the model reduct for procedure symbols (which ensures minimality) and from Lemma 12.2.3.

3. e is a restricted sort generation constraint.

The satisfaction condition is obvious.

12.3 VSE Refinement as an Institution Comorphism

By contrast with the behavioral refinement of CASL, the VSE specification language supports a refinement approach based on explicit submodels and congruences [Reif, 1992], an idea that dates back to Hoare [Hoare, 1972]. This more specific and simpler approach has been successfully applied in practice, and moreover, it is linked with a code generation mechanism. Hence, integrating this approach into HETS brings considerable advantages.

VSE's refinements associate an abstract data type specification, called the *export specification* of the refinement, with an implementation. The implementation is

based on another theory, called the *import specification* and contains several functional procedures written in an imperative language. These procedures use the functions and predicates of the import specifications. A so called *mapping* relates each sort of the export specification to a sort of the import specification, while the functions and procedures are mapped to procedures in the import specification.

A refinement describes the construction of a model for the signature of the export specification (export model) from a model of the import specification (import model). The functions and predicates are interpreted by the computations of the procedures. The elements of the carrier sets of the export model are constructed from the carrier sets of the import model. The implementations are allowed to represent a single value in the export specification by several values of the import specifications. For example, when implementing sets by lists, a set might be represented by any list containing all elements of the set in any order. Furthermore, VSE does not require that all values of a sort in the import specification really represent a value of the export specification. In the example below where we will implement natural numbers by binary words, we will exclude words with leading zeroes. In order to describe the construction of the carrier sets, the refinement contains two additional procedures for each sort: a procedure defining a *congruence* relation and a procedure defining a *restriction*. The restriction terminates on all elements that represent export specification values. The congruence relation determines the equivalence classes that represent the elements of the export model.

We can express this formally as in the following definition:

Definition 12.3.1 *Let SP be a $CFOL^=$ -specification and SP' a $CDyn^=$ -specification. Then SP' is a refinement of SP , denoted $SP \rightsquigarrow_{VSE} SP'$, if for all $M \in Mod^{CDyn^=}(SP')$, $\langle M \rangle / \equiv \in Mod^{CFOL^=}(SP)$, where $\langle M \rangle / \equiv$ denotes the model obtained from M by restricting the elements of each sort according to the restriction procedures and taking the quotient to the congruence relation.*⁵

Note that the definition is given in semantic terms. The VSE system generates proof obligations that are sufficient for guaranteeing that a $CFOL^=$ -specification is indeed a refinement of a $CDyn^=$ -specification.

When integrating VSE and its notion of refinement into HETS, a naive approach would extend HETS with a new notion of *restriction-quotient refinement link* in HETS, and would extend both the HETS motherboard and the expansion slot specification in a way that makes it possible to deal with such refinement links. VSE easily could be turned into an expansion card that is able to prove these refinement links.

However, this approach has a severe disadvantage: the specification of expansion slots needs to be extended! If we did this for every tool that is newly integrated into HETS (and every tool comes with its own special features), we would quickly arrive at a very large and unmanageable expansion slot specification.

⁵For a definition of quotients of first-order models, see [Sannella and Tarlecki, 2012].

Fortunately, the heterogeneity of HETS offers a better solution: we can encode VSE refinement as ordinary refinement in HETS, with the help of an institution comorphism that does the actually restriction-quotient construction. With this approach, only the HETS logic graph needs to be extended by a logic and a comorphism; actually, we will see that two comorphisms are necessary. That is, we add two further expansion cards doing the work, while the logic-independent part of HETS, i.e. the motherboard and the expansion slot specification, can be left untouched!

12.3.1 The Refinement Comorphism

We model the refinement notion of VSE by a comorphism from the CASL institution $CFOL^=$ to the VSE institution $CDyn^=$. The intuition behind it can be summarized as follows. At the level of signatures, for each sort we need to introduce procedure symbols for the equality relation and for the restriction formula together with axioms specifying their expected behavior, while for function and predicate symbols, we need to introduce procedure symbols for their implementations. For all these symbols, we assign no procedure definition but rather leave them loosely specified; in this way, the choice of a possible implementation is not restricted. The sentence translation is based on translation of terms into programs implementing the representation of the term. The model reduct performs the submodel/quotient construction, leaving out the values that do not satisfy the restriction formula and quotienting by the congruence generated by the equality procedure.

We now define the simple theoroidal comorphism $CASL2VSERefine : CFOL^= \rightarrow CDyn^=$. Each CASL signature $\Sigma = (S, F, P)$ is mapped to the $CDyn^=$ theory $((S, \emptyset, \emptyset, PR), E)$, denoted $\Phi(\Sigma)$. PR contains (1) for each sort s , a symbol $restr_s \in PR_{[s], []}$ for the *restriction* on the sort and a symbol $eq_s \in PR_{[s, s], [Boolean]}$ for the *equality* on the sort and (2) for each function symbol $f : w \rightarrow s \in F_{w, s}$, a symbol $gn_f : w \rightarrow s \in PR_{w, [s]}$ and for each predicate symbol $p : w \in P_w$, a symbol $gn_p : w \rightarrow [Boolean] \in PR_{w, [Boolean]}$.

The set of axioms E contains sentences saying that for each sort s , (1) eq_s is a congruence and it terminates for inputs satisfying the restriction and (2) the procedures that implement functions/predicates terminate for inputs satisfying the restriction and their results also satisfy the restriction. These properties are to be proven when providing an actual implementation. The general pattern of the translation is presented in Fig. 12.2, which gives the symbols and the sentences introduced in the resulting VSE theory for each symbol of the CASL theory that is translated. To improve readability, we only considered the case of unary function/predicate symbols; the generalization to symbols of arbitrary arity is obvious. Moreover, \top stands for *True*, B for *Boolean* and the restriction $restr_s$ is abbreviated r_s .

A CASL signature morphism $\sigma : \Sigma \rightarrow \Sigma'$ is mapped to the $CDyn^=$ morphism $\Phi(\sigma) : \Phi(\Sigma) \rightarrow \Phi(\Sigma')$ which works like σ on sorts and procedure symbols corresponding to function/predicate symbols in Σ and for each sort s of Σ maps eq_s to $eq_{\sigma(s)}$ and $restr_s$ to $restr_{\sigma(s)}$.

CASL	VSE	VSE sentences
sort s	$\text{sort } s$ $eq_s \in PR_{[s,s],[B]}$ $r_s \in PR_{[s],[\]}$	$\langle r_s(x) \rangle \top \wedge \langle r_s(y) \rangle \top \Rightarrow \langle eq_s(x, y; e) \rangle \top$
		$\langle r_s(x) \rangle \top \Rightarrow \langle eq_s(x, x; e) \rangle e = \top$
		$\langle r_s(x) \rangle \top \wedge \langle r_s(y) \rangle \top \wedge \langle eq_s(x, y; e) \rangle e = \top \Rightarrow \langle eq_s(y, x; e) \rangle e = \top$
		$\langle r_s(x) \rangle \top \wedge \langle r_s(y) \rangle \top \wedge \langle r_s(z) \rangle \top \wedge \langle eq_s(x, y; e) \rangle e = \top \wedge \langle eq_s(y, z; e) \rangle e = \top \Rightarrow \langle eq_s(x, z; e) \rangle e = \top$
$f \in F_{s \rightarrow t}$	$gn_f \in PR_{[s],[t]}$	$\langle r_s(x) \rangle \top \wedge \langle r_s(y) \rangle \top \wedge \langle eq_s(x, y; e) \rangle e = \top \Rightarrow \langle y1 := gn_f(x) \rangle \langle y2 := gn_f(y) \rangle \langle eq_t(y1, y2; e) \rangle e = \top$
		$\langle r_s(x) \rangle \top \Rightarrow \langle gn_f(x; y) \rangle \langle r_t(y) \rangle \top$
$p \in P_s$	$gn_p \in PR_{[s],[B]}$	$\langle r_s(x) \rangle \top \wedge \langle r_s(y) \rangle \top \wedge \langle eq_s(x, y; e) \rangle e = \top \Rightarrow \langle gn_p(x; r1) \rangle \langle gn_p(y; r2) \rangle r1 = r2$
		$\langle r_s(x) \rangle \top \Rightarrow \langle gn_p(x; e) \rangle \top$

Figure 12.2: Summary of the signature translation part of the comorphism $CASL2VSERefine$ (for simplicity, only unary symbols are shown).

Given a CASL signature $\Sigma = (S, F, P)$ and a model M' of its translation $\Phi(\Sigma) = ((S, \emptyset, \emptyset, PR), E)$, we define the translation of M' to an (S, F, P) -model, denoted $M = \beta_\Sigma(M')$. The interpretation of a sort s in M is constructed in two steps. First we take the subset $M_{restr_s} \subseteq M'_s$ of elements, for which the restriction predicate holds. Then we take the quotient M_{restr_s}/\equiv according to the congruence relation \equiv defined by eq_s , such that for all $a, b \in M'_s$, $a \equiv b$ is equivalent to $M', t \models \langle eq_s(x_1, x_2; y) \rangle y = true$ whenever t is a state such that $t(x_1) = a$ and $t(x_2) = b$. For each function symbol f , we define the value of M_f in the arguments a_1, \dots, a_n to be the value returned by the call of procedure M'_{gn_f} on inputs a_1, \dots, a_n , that is $M_f(a_1, \dots, a_n) = b$ if and only if $M', t \models \langle gn_f(x_1, \dots, x_n; y) \rangle y = z$ when t is a state such that $t(x_i) = a_i$ for any $i = 1, \dots, n$ and $t(z) = b$. Axioms (1) and (2) in E ensure that M_f is total and well-defined. Similarly and using the same notations, for each predicate symbol p , $M_p(a_1, \dots, a_n)$ holds iff $M', t \models \langle gn_p(x_1, \dots, x_n; y) \rangle y = true$.

Proposition 12.3.2 *The model translation is natural.*

Proof. Follows easily from definition of the model translation and the definition of model reducts. \square

Sentence translation is based on translation of terms into programs that compute the representation of the term. Basically, each function application is translated to a procedure call of the implementing procedure, and new output variables are introduced:

- a variable x is mapped to $x := x$, where the left-hand side x is the output variable and the right-hand side x is the logical variable;
- a constant c is mapped to $gn_c(; y)$, where gn_c is the procedure implementing the constant and y is a new output variable;
- a term $f(t_1, \dots, t_n)$ is mapped to $\alpha_1; \dots; \alpha_n; a := gn_f(y_1, \dots, y_n)$, where α_i is the translation of t_i with the output variable y_i and a is a new output variable.

Then the sentence translation is defined inductively:

- an equation $t_1 = t_2$ is translated to

$$\langle \alpha_1 \rangle; \langle \alpha_2 \rangle; \langle eq_s(y_1, y_2; y) \rangle y = true$$

where α_i is the translation of the term t_i , with the output variable y_i

- a predicate application $p(t_1, \dots, t_n)$ is translated to

$$\langle \alpha_1 \rangle \dots \langle \alpha_n \rangle \langle gn_p(y_1, \dots, y_n; y) \rangle y = true$$

where α_i is the translation of the term t_i , with the output variable y_i

- Boolean connectives of formulas are translated into the same connections of their translated formulas;
- for universally and existentially qualified formulas one also has to make sure that the bound variables are assigned a value that satisfies the restriction: e.g. $\forall x : s.e$ gets translated to $\forall x : s. \langle restr_s(x) \rangle true \Rightarrow \alpha(e)$, where we denoted with $\alpha(e)$ the translation of e .

An example of how a CASL sentence is translated along the *CASL2VSERefine* comorphism will be introduced in the next section in Fig. 12.4.

Sort generation constraints are translated to restricted sort generation constraints over implementing procedures. For example, assume we have in the abstract specification of natural numbers a sort generation constraint:

$$generated\ type\ nat ::= 0 \mid suc\ (nat)$$

Then in the VSE theory resulting from translation along comorphism, the restricted sort generation constraint

$$\mathbf{generated\ type\ } nat ::= gn_0 \mid gn_suc(nat) \mathbf{\ restricted\ by\ } restr_nat .$$

is introduced, where gn_0 and gn_suc are the procedures implementing the constructors and $restr_nat$ is the restriction procedure symbol on sort nat .

Proposition 12.3.3 *The sentence translation is natural.*

Proof. Follows easily by induction on sentences and by noticing that for any CASL signature morphism $\sigma : \Sigma \rightarrow \Sigma'$ and any Σ -term t we have that the program computing the representation of $\sigma(t)$ is the σ -image of the program computing the representation of t . \square

Lemma 12.3.4 *Let Σ be a $CFOL^=$ -signature and let M' be a model of the theory $\Phi(\Sigma)$. Denoting $M = \beta_\Sigma(M')$ we have that for any Σ -term t and any state q for $\Phi(\Sigma)$ and M' , $M_t = a$ iff $M', q \models \langle \alpha_\Sigma(t) \rangle y = z$, where y the output variable of $\alpha_\Sigma(t)$ and z is a variable such that $q(z) = a$.*

Proof. Follows by induction on the structure of the term t . \square

Theorem 12.3.5 *The satisfaction condition for the comorphism $CASL2VSERefine$ holds.*

Proof. Follows by induction on the structure of the sentences and making use of Lemma 12.3.4. \square

This construction follows very faithfully the steps of the refinement method of VSE, as described above. The export specification of VSE is a first-order specification that we can translate along the comorphism $CASL2VSERefine$ to generate the same kind of proof obligations that VSE would generate to prove correctness of a VSE refinement. The difference is that now they are built using abstract (i.e. loose) procedure names and actual implementations are to be later plugged in by means of a refinement (in HETS) along a signature morphism which corresponds to the VSE mapping, with the exception that instead of pairing export specification symbols with implementations, the morphisms rather pairs abstract procedures with implementations. Moreover, the correctness of the HETS refinement ensures us that a model of the implementation reduces along the signature morphism to a model of the translation of the original export specification, that we can further translate along the comorphism to obtain a model of the export specification. Thus we achieve that the model semantics of the refinement in VSE [VSE, 1997] and of the refinement expressed using the comorphism $CASL2VSERefine$ coincide.

Definition 12.3.6 *Let I and J be two institutions, $\rho = (\varphi, \alpha, \beta) : I \rightarrow J$ be an institution comorphism. Let SP be a I specification and SP' be a J specification. We say that SP' is a heterogeneous refinement of SP along ρ if for each $M \in Mod^J(SP')$, $\beta(M) \in Mod^I(SP)$.*

Our result can be formulated as follows:

Theorem 12.3.7 *Let SP be a CASL specification and SP' a VSE specification. Then $SP \sim_{VSE} SP'$ iff SP' is a heterogeneous refinement of SP along $CASL2VSERefine$.*

We can now show that the proof calculus for heterogeneous development graphs, combined with the VSE prover, can be used for discharging refinement proof obligations in a sound way.

The following two lemmas follow easily:

Lemma 12.3.8 *The institution $CDyn^=$ has the amalgamation property.*

Proof idea: Similar to the proof for first-order logic, see for example [Diaconescu, 2008].

Lemma 12.3.9 *The comorphism $CASL2VSERefine$ admits model expansion.*

Proof. For any $CFOL^=$ signature Σ and each Σ -model M , we build a $\Phi(\Sigma)$ -model by interpreting sorts s as M_s , functions gn_f like M_f , predicates gn_p as M_p , the equality as the set-theoretical equality and the restriction as always returning *true*. It is easy to see that the model such built satisfies the axioms of $\Phi(\Sigma)$ and it reduces via β_Σ to M . \square

In VSE we do not have hiding as a structuring operation, and therefore all specifications are flattenable (see e.g. [Sannella and Tarlecki, 2012]). The following corollary follows then directly from Lemma 12.3.9 and a result from [Mossakowski, 2002b].

Corollary 12.3.10 *The comorphism $CASL2VSERefine$ admits borrowing of entailment and of refinement.*

Unfortunately, we cannot expect completeness here, because first-order dynamic logic is not finitely axiomatisable [Blackburn et al., 2006].

12.3.2 Structuring in Context of Refinement

Consider a refinement from an abstract to a refined specification where a theory of a library (e.g. the natural numbers) or a parameter theory that will be instantiated later occurs both in the abstract and the refined specification. Such common import specifications should not be refined, but rather kept identically — and this is indeed the case in VSE.⁶

To handle this situation in the present context, the import of a first-order specification into a dynamic logic specification is not done along the trivial inclusion comorphism from $CFOL^=$ to $CDyn^=$ — this would mean that the operations of the import need to be implemented as procedures. Instead, we define a comorphism $CASL2VSEImport : CFOL^= \rightarrow CDyn^=$, which, besides keeping the first-order part, will introduce for the symbols of the import specification new procedure symbols, similarly to $CASL2VSERefine$. Namely each $CFOL^=$ signature (S, F, P) is translated to the $CDyn^=$ theory $((S, F, P, PR), E)$ where PR is the same as in the definition of the translation of (S, F, P) along $CASL2VSERefine$ and E contains the following types of sentences:

⁶This resembles a bit the notion of imports of parameterized specifications in CASL [Mosses, 2004], where the import is shared between formal and actual parameter and is kept identically.

- for each sort $s \in S$, sentences giving the implementations for the restriction and the equality of s , as follows:

```
PROCEDURE  $restr_s(x)$ 
BEGIN SKIP END;
```

and respectively

```
FUNCTION  $eq_s(x, y)$ 
BEGIN IF  $x = y$  THEN RETURN True
ELSE RETURN False FI END;
```

with the intuitive meaning that no element of the sort is restricted and the equality on the sort is defined as the (meta-)equality on the interpretation of the sort;

- for each operation symbol $f \in F_{s \rightarrow t}$, a sentence giving the implementation of the corresponding procedure symbol $gn_f \in PR_{[s],[t]}$:

```
FUNCTION  $gn\_f(x)$ 
BEGIN DECLARE  $y : t := f(x)$ ; RETURN  $y$  END;
```

which means that the implementation of the functional procedure for f returns as result exactly the value $f(a)$ for each input a ;

- for each predicate symbol $p \in P_w$, a sentence giving implementation of the corresponding procedure symbol $gn_p \in PR_{[w],[Boolean]}$:

```
FUNCTION  $gn\_p(x)$ 
BEGIN DECLARE  $y : Boolean := p(x)$ ; RETURN  $y$  END;
```

again with the meaning that the functional procedure gn_p is implemented as returning true only on those inputs that make the predicate p hold.

The result of choosing these implementations is that the sorts are not restricted, the congruence on each sort is simply the equality and the functional procedures introduced for operation/predicate symbols have the same behavior as the original symbols, i.e. give the same results on same inputs. Moreover, the signature morphisms translation of the comorphism $CASL2VSEImport$ is the straightforward one, the translation of CASL sentences along the comorphism is simply the identity, and the models can be reduced in an obvious way by simply forgetting the interpretations of procedure symbols. The satisfaction condition of the comorphism follows immediately.

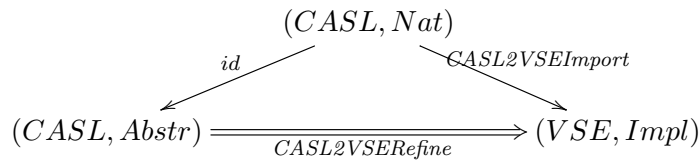


Figure 12.3: Common import.

For example, let us consider the situation in Fig. 12.3, where the natural numbers are imported both in the abstract and the concrete specification and the (heterogeneous) refinement link is represented by the double arrow. The label *CASL2VSEImport* on the right arrow indicates that *Nat* is translated via the import comorphism before being imported in *Impl*. We can assume for simplicity that *Nat* has only a sort *nat* and then in *Impl* we have procedure symbols for identification and restriction on *nat*, together with procedure definitions saying that no element is restricted and the identification procedure is simply equality.

On the other side, when *Abstr* is translated along the refinement comorphism to $CDyn^=$, no distinction between the sorts defined in *Abstr* and the imported ones is made, so in the resulting translated theory we will have symbols for the restriction on sort *nat* and for identification. These symbols are then mapped identically by the $CDyn^=$ -signature morphism that labels the refinement and since in *Impl* no restriction and no identification on *nat* is made, the quotient on *nat* is trivial. For any function/predicate symbols from *Nat* we would get the same behavior: the default implementations provided by *CASL2VSEImport* act only as wrappers for functions/procedures, without changing their values, and therefore the imported specification symbols are kept identically.

12.4 Example: Implementing natural numbers by binary words

As an example, we present the implementation of natural numbers as lists of binary digits, slightly abridged from [Reif, 1992].⁷ The abstract CASL specification NATS is the usual specification of natural numbers with 0, successor and addition with the Peano axioms.

```
spec NATS =
  free type nats ::= zero_n | succ_n(nats)
  op   zero_n : nats
  op   succ_n : nats → nats
  op   prdc_n : nats → nats
  op   add_n : nats × nats → nats
  vars m, n : nats
  • prdc_n(zero_n) = zero_n
  • prdc_n(succ_n(m)) = m
  • add_n(m, zero_n) = m
  • add_n(m, succ_n(n)) = succ_n(add_n(m, n))
```

The predecessor function is defined to take *zero_n* to *zero_n*; this is because in VSE there are no partial functions. In Fig. 12.4⁸, we present a fragment of the

⁷ The complete example can be found at https://svn-agbkb.informatik.uni-bremen.de/Hets-lib/trunk/Refinement/natbin_refine.het.

⁸HETS uses $\langle \alpha \rangle \varphi$ as input syntax for $\langle \alpha \rangle \varphi$.

theory obtained by translating `NATS` along the comorphism `CASL2VSERefine`: the resulting signature and the translation of the first axiom - the other three translated axioms and the sentences introduced by the comorphism are similar.

```

sort  nats
PROCEDURES
  gn_add_n : IN nats, IN nats → nats;
  gn_eq_nats : IN nats, IN nats → Boolean;
  gn_prdc_n : IN nats → nats;
  gn_restr_nats : IN nats;
  gn_succ_n : IN nats → nats;
  gn_zero_n : → nats
∀ gn_x0 : nats; gn_x1 : nats; gn_x2 : nats; gn_x3 : Boolean
• <:gn_x1 := gn_zero_n;
  gn_x0 := gn_prdc_n(gn_x1);
  gn_x2 := gn_zero_n;
  gn_x3 := gn_eq_nats(gn_x0, gn_x2):>
  gn_x3 = (op True : Boolean)
...

```

Figure 12.4: Natural numbers translated along the comorphism `CASL2VSERefine`.

The VSE implementation, `NATS-IMPL` (Fig. 12.5), provides procedures for the implementation of natural numbers as binary words, which are imported as data part along `CASL2VSEImport` from the CASL specification `BIN` (omitted here). We illustrate the way the procedures are written with the example of the restriction procedure, `nlz`, which terminates whenever the given argument has no leading zeros. The implementation of the other procedures is similar and therefore omitted. Notice that the equality is in this case simply the equality on binary words.

We now have to express that binary words, restricted to those with non-leading zeros, represent a refinement of natural numbers. We can express this using the refinement language introduced in Chapter 11, with the Grothendieck institution of HETS as underlying institution, to allow capturing heterogeneous refinements. Fig. 12.6 presents this refinement, where we record that each symbol of `NATS` is implemented by the corresponding procedure in the symbol mapping of the view.

In Fig. 12.7, we present some of the proof obligations introduced by the refinement. They are translations of the sentences of the theory presented in Fig. 12.4 along the signature morphism induced by the symbol map of the refinement. The first two sentences are introduced by the signature translation of the comorphism and state that (1) equality terminates on inputs for which the restriction formula `nlz` holds and (2) the procedure implementing addition, `i_add`, terminates for valid inputs and the result is again valid. Also the translation of an axiom of `NATS` along the comorphism `CASL2VSERefine` is presented. The resulting development graph is displayed in Fig. 12.8, where the double arrows correspond to translations along

```

spec NATS_IMPL =
  BIN with logic → CASL2VSEImport
then PROCEDURES
  hnlz : IN bin; nlz : IN bin; i_badd : IN bin, IN bin, OUT bin, OUT bin;
  i_add : IN bin, IN bin → bin; i_prdc : IN bin → bin;
  i_succ : IN bin → bin; i_zero : → bin; eq : IN bin, IN bin → Boolean
  • DEFPROCS
    PROCEDURE hnlz(x)
      BEGIN
        IF x = b_zero THEN ABORT
        ELSE IF x = b_one THEN SKIP ELSE hnlz(pop(x)) FI
      FI
    END;
    PROCEDURE nlz(x)
      BEGIN IF x = b_zero THEN SKIP ELSE hnlz(x) FI END
  DEFPROCSSEND
%% ...

```

Figure 12.5: Implementation using lists of binary digits.

```

refinement BINARY_ARITH =
  NATS refined via
    logic → CASL2VSERefine,
    nats ↦ bin, gn_restr_nats ↦ nlz, gn_eq_nats ↦ eq,
    gn_zero_n ↦ i_zero, gn_succ_n ↦ i_succ,
    gn_prdc_n ↦ i_prdc, gn_add_n ↦ i_add
to NATS_IMPL

```

Figure 12.6: Natural numbers as binary words.

```

%% Proof obligations introduced by the refinement
%% equality procedure terminates on valid inputs
 $\forall gn\_x, gn\_y : bin \bullet \langle :nlz(gn\_x) \rangle :> true \wedge \langle :nlz(gn\_y) \rangle :> true$ 
 $\Rightarrow \langle :gn\_b := eq(gn\_x, gn\_y) \rangle :> true$ 
%% procedure implementing addition terminates and gives valid results on valid inputs
 $\forall gn\_x1, gn\_x2 : bin \bullet \langle :nlz(gn\_x1) \rangle :> true \wedge \langle :nlz(gn\_x2) \rangle :> true$ 
 $\Rightarrow \langle :gn\_x := i\_add(gn\_x1, gn\_x2) \rangle :> \langle :nlz(gn\_x) \rangle :> true$ 
%% translation of : forall m : nats . add_n(m, zero_n) = m
 $\forall gn\_x0, gn\_x1, gn\_x2, gn\_x3 : bin; gn\_x4 : Boolean;$ 
 $m : bin$ 
 $\bullet \langle :nlz(m) \rangle :> true$ 
 $\Rightarrow \langle :gn\_x1 := m ;$ 
 $gn\_x2 := i\_zero;$ 
 $gn\_x0 := i\_add(gn\_x1, gn\_x2);$ 
 $gn\_x3 := m;$ 
 $gn\_x4 := eq(gn\_x0, gn\_x3) \rangle :>$ 
 $gn\_x4 = (op True : Boolean)$ 

```

Figure 12.7: Generated proof obligations

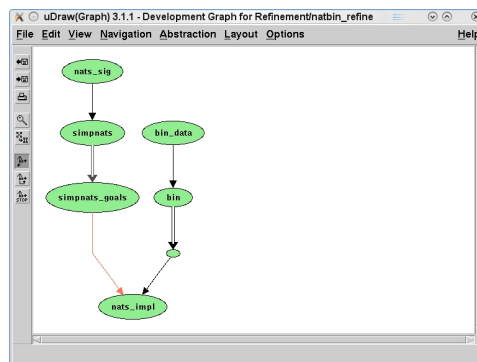


Figure 12.8: The development graph of natural numbers example

comorphisms and the red arrow is introduced by the refinement.

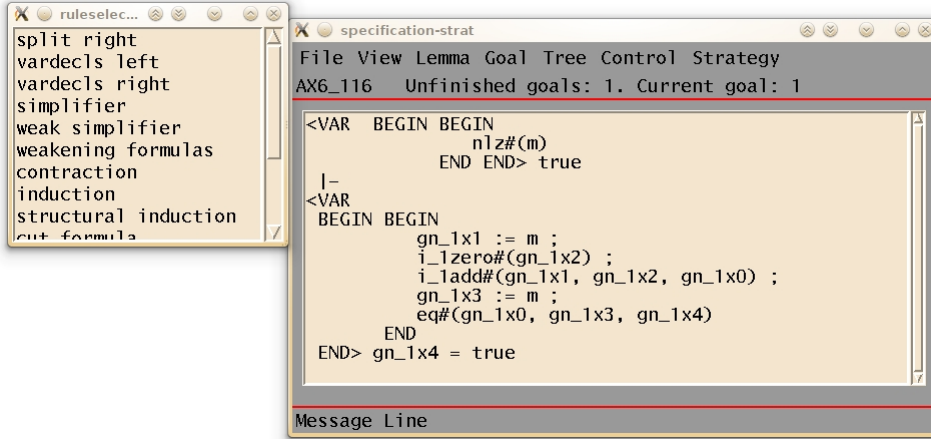


Figure 12.9: A sample proof goal in VSE

The proof obligations together with all axioms can be handed over to the VSE prover; the user is presented with the interactive prover interface, where the current proof state can be inspected and proofs of obligations or lemmas can be started. Fig. 12.9 shows the situation after starting the proof for the obligation resulting from the axiom describing zero as the neutral element with respect to addition. There is a window containing the current goal and another window with a list of applicable rules to choose from.

The prover uses a sequent calculus. Therefore the goals have the form of sequents $e_1, e_2, \dots, e_n \vdash e'_1, e'_2, \dots, e'_m$ meaning that under the assumption of e_1, e_2, \dots, e_n one of the formulas e'_1, e'_2, \dots , or e'_m holds.

The user could now complete the proof by selecting rules by hand. For this kind of dynamic logic goals rules for each program construct are available. For example for a conditional **if** ε **then** α **else** β **fi** we have the rule

$$\frac{\Gamma, \varepsilon \vdash \langle \alpha \rangle e, \Delta \quad \Gamma, \neg \varepsilon \vdash \langle \beta \rangle e, \Delta}{\Gamma \vdash \langle \text{if } \varepsilon \text{ then } \alpha \text{ else } \beta \text{ fi} \rangle e, \Delta}$$

where Γ and Δ are sequences of formulas. Applying this rule would generate two new goals, one assuming the condition ε holds which allows us to replace the conditional with α , and the other one assuming $\neg \varepsilon$.

The rule for an assignment statement $x := \tau$ will change the sequent in a way that it reflects the state after the assignment. It will remove all formulas where x occurs freely and add the equation $x = \tau$. In the following rule Γ' resp. Δ' are obtained from Γ resp. Δ by removing all formulas with free occurrences of the variable x :

$$\frac{\Gamma', x = \tau \vdash e, \Delta'}{\Gamma \vdash \langle x := \tau \rangle e, \Delta}$$

A simplifier is run after each rule application. When appropriate, it will apply a substitution $x = \tau$ on e and remove the equation $x = \tau$. For example, starting from the goal in Fig. 12.9, the user would soon want to get rid of the assignment $gn_1x1 := m$, which results in the following new goal:

```
<nlz#(m)> true
|-
<i_1zero#(gn_1x2)>
  <i_1add#(m, gn_1x2, gn_1x0)>
    <gn_1x3 := m>
    <eq#(gn_1x0, gn_1x3, gn_1x4)> gn_1x4 = true
```

Next the call of procedure i_1zero has to be dealt with. There is a rule which allows to unwind procedure calls. In this case it will yield the following new goal:

```
<nlz#(m)> true
|-
<VAR BEGIN BEGIN res-i_1zero := b_1zero
      END END>
  <i_1add#(m, res-i_1zero, gn_1x0)>
    <gn_1x3 := m>
    <eq#(gn_1x0, gn_1x3, gn_1x4)> gn_1x4 = true
```

The proof then would continue always choosing rules for the first top level program construct. The user will rather activate the heuristics for that, and thus most of the remaining proof is done automatically. As these heuristics are mainly driven by a program appearing in one of the formulas and the result looks like executing the program with symbolic terms instead of values, it is called *symbolic execution*.

In general VSE performs a heuristic loop, which means that after each rule application it tries to apply heuristics from a given list of heuristics the user has chosen. In case all heuristics should fail, there is also a last resort heuristic which allows the user to select a rule from the set of all applicable rules.

Finally, a proof tree as shown in Fig. 12.10 results, where each goal is shown as a node and each rule application lets the tree grow upwards.

A more involved example is the proof obligation that will show that the procedure i_add , if applied to well-formed input arguments, terminates and produces a well-formed result (in the sense of the restriction procedure):

```
<nlz#(gn_1x1)> true, <nlz#(gn_1x2)> true
|-
< i_1add#(gn_1x1, gn_1x2, gn_1x)>
  < nlz#(gn_1x)> true
```

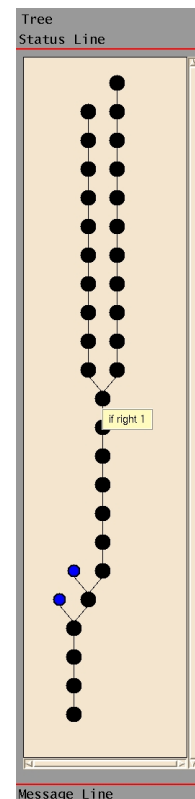


Figure 12.10:
VSE proof tree

Many proof steps still can be done by symbolic execution. However, as i_add is recursive this could fail to complete the proof and lead to an infinite loop instead. To prevent this, an induction proof is required, in this case structural induction on the first input argument i_add . The induction hypothesis can then be used for recursive calls of i_add . The proof should also be made more concise by avoiding to unwind the i_succ calls occurring in i_add . Instead these calls should be handled by using the similar proof obligation

```
<nlz#(gn_1x1)> true
|-
< i_1succ#(gn_1x1, gn_1x)> < nlz#(gn_1x)> true
```

as a lemma.

Most of the proof obligations for this example can be treated similar to the two obligations we have discussed. After finishing the proof with the VSE prover, HETS is informed about the obligations that have been completed.

The two comorphisms have been implemented and are part of HETS; the VSE tool is also going to become available under public license. Provided VSE is installed, the example can be fully checked in HETS.

12.5 Conclusions and future work

We have integrated VSE's mechanism of refining abstract specifications into procedural implementations into HETS. Via a new logic and two logic translations, one of them doing the usual restriction-quotient construction, we could avoid entirely the introduction of new types of "refinement links" into HETS, but rather could re-use the present machinery of heterogeneous development graphs and thus demonstrate its flexibility. Visually spoken, we could avoid extending the HETS motherboard and expansion slot specification, but rather just construct several expansion cards related to VSE and plug them into the HETS motherboard.

However, there is a point when it actually makes sense to enhance the expansion slot specification. Currently, it is based on the assumption that expansion cards (aka theorem provers) can only handle flat unstructured theories. However, VSE can also handle structured theories, and takes advantage of the structuring during proof construction. Hence, we plan to extend the expansion slot specification in a way that allows the transmission (between HETS and VSE) of whole acyclic directed development graphs of theories with connecting definition links, reflecting the import hierarchy. We expect to use this enhancement of the expansion slot specification also for other theorem provers supporting structured theories, like Isabelle.

Another direction of future work will try to exploit synergy effects between VSE and HETS e.g. by using automatic provers like SPASS (which are now available through the integration) during some sample VSE refinement proofs. The refinement method could also be extended from first-order logic to the richer language CASL, covering also features like subsorting and partial functions.

Acknowledgement. The results of this chapter have been published in [Code-scu et al., 2012]. I have participated to the definition of the institution of dynamic logic (Sec. 12.2). The integration of VSE and HETS was mainly done by C. Maeder and B. Langenstein. Moreover, I defined and implemented in HETS the two comorphisms from CASL to VSE (Sec. 12.3). The VSE proof in Sec. 12.4 is due to B. Langenstein.

Part IV

Final Remarks

Conclusions and Future Work

Contents

13.1 Summary	223
13.2 Future Work	224

In the final chapter of this thesis we give a brief summary of the results and discuss directions of future work.

13.1 Summary

The central objective of this work is to further develop the Heterogeneous Tool Set HETS, a tool for heterogeneous specification which interfaces a number of formalisms and their tools.

In Part II of this thesis we report on a number of recent extensions of HETS.

We have integrated in HETS declarative specifications of logics in a logical framework like Edinburgh LF, Isabelle or Maude. This adds a new dimension to HETS, in the sense that the users of the tool can now extend it with a new logic, without having to understand the details of its implementation.

We have then taken a look at a number of logic translations in the graph of logics of HETS that fall outside existing definition of institution comorphisms, which is used as a formalization of logic translation in HETS. We have therefore developed a generalization of this concept and studied the impact of the proposed generalization to heterogeneous specification and heterogeneous proofs.

We have designed and implemented an algorithm for computing approximations of colimits for heterogeneous diagrams. This is of particular importance for dealing with hiding in the development graph calculus of HETS. The corresponding rules of this calculus are as a result supported as well by HETS.

We have furthermore added Maude as a new logic in HETS, providing thus not only one more framework for specifying logics but also a state-of-the-art term rewriting system. Since Maude is based heavily on initial semantics, using a special notion of freeness to model it, we have developed an non-disruptive encoding of Maude freeness, in terms of the existing links in development graphs. This is followed by a normalization of CASL free definition links that facilitates proof support for Maude.

In Part III of this work we concentrated on supporting the CASL refinement language in HETS.

The CASL refinement language has been developed as an extension of the existing architectural level of CASL. We start therefore with an enhancement of the semantics of architectural specifications which also provides a simplification in the verification and refinement of units with imports.

We then move to the entire refinement language. We have complemented the language with an explicit notion of refinement trees, which allows to visualize the structure of the development and provides access points for the logical properties of refinement specifications, such as consistency. We moreover equip the refinement language with a proof calculus for correctness and study its soundness and completeness. The latter is obtained rather easily for the architectural sublanguage, but can only be achieved by exploiting information about the choice of a certain implementation in the general case. Finally, we derive a consistency check calculus for refinement specifications and prove its soundness and completeness. From an implementation perspective, the refinement language, the proof calculus for refinements and the consistency calculus are now supported by HETS.

Finally, we integrated the Verification Support Environment VSE in HETS, together with its notion of refinement that has similarities to the observational refinement of CASL. We have taken advantage of the support for heterogeneity in HETS to model the VSE refinement as an institution comorphism. This allows us to use the simple refinement language of HETS for writing down VSE refinements. As a result, we do not have to extend the development graph calculus with a special type of VSE refinement link, but we can rather re-use the existing concepts.

13.2 Future Work

Future work is largely discussed at the end of each corresponding chapter, as well as comparison with related work. Therefore, in the following we will just present a brief overview of the most important aspects.

Firstly, the examples presented in this thesis are of rather medium size. An important scalability test is to develop more complex case studies of software development with HETS, using the refinement language to record the phases of development. Towards this purpose, it is important to add full support for heterogeneity in the refinement language of HETS, including heterogeneous architectural specifications.

Another interesting challenge is to support observational refinement in HETS in a more general case than the one provided by VSE. [Bidoit et al., 2008] give a solution for the case when the underlying institution is CASL; the next step would be to generalize the construction to the heterogeneous case and to implement this generalization in HETS.

Finally, in Chap. 6 we presented an extension of HETS that makes it possible to add new logics in a declarative way. Further work is required to enhance support for these logics. One drawback of the approach is that currently the syntax of the new logics is inherited from the framework. We are looking at ways of generating

a concrete syntax that is logic specific, possibly using the names of the declaration patterns as keywords of the language. This should be followed by a transformation from the syntax of the declared logic to the syntax of a hard-coded logic in HETS, such that the declared logic is also supported by tools, possibly via institution comorphisms. As a final step, the various tool interfaces of HETS should also be made more declarative, such that HETS logics specified in a logical framework can be directly connected to theorem provers and other tools, instead of using a comorphism into a hard-coded logic. Then, in the long run, it will be possible to entirely replace the hard-coded logics with declarative logic specifications in the LATIN metaframework — and only the latter needs to be hard-coded into HETS.

Bibliography

- Spezifikationsprache VSE-SL, 1997. Part of the VSE documentation. (Cited on page 210.)
- OWL 2 web ontology language: Document overview. W3C recommendation, World Wide Web Consortium (W3C), October 2009. URL <http://www.w3.org/TR/2009/REC-owl2-overview-20091027/>. (Cited on page 27.)
- J. Abrial, E. Börger, and H. Langmaack, editors. *Formal Methods for Industrial Applications, Specifying and Programming the Steam Boiler Control (the book grew out of a Dagstuhl Seminar, June 1995)*, volume 1165 of *Lecture Notes in Computer Science*, 1996. Springer. ISBN 3-540-61929-1. (Cited on page 62.)
- J. Adámek, H. Herrlich, and G. Strecker. *Abstract and Concrete Categories*. Wiley, New York, 1990. (Cited on page 33.)
- E. Astesiano, H.-J. Kreowski, and B. Krieg-Brückner, editors. *Algebraic Foundations of System Specification*. IFIP State-of-the-Art Reports. Springer-Verlag; D-69121 Heidelberg, Germany; <http://www.springer.de>, 2000. (Cited on pages 12 and 15.)
- S. Autexier, D. Hutter, B. Langenstein, H. Mantel, G. Rock, A. Schairer, W. Stephan, R. Vogt, and A. Wolpers. VSE: Formal methods meet industrial needs. *International Journal on Software Tools for Technology Transfer, Special issue on Mechanized Theorem Proving for Technology*, 3(1), september 2000. (Cited on pages 18, 195 and 197.)
- A. Avron, F. Honsell, M. Miculan, and C. Paravano. Encoding modal logics in logical frameworks. *Studia Logica*, 60(1):161–208, 1998. (Cited on page 80.)
- F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, and P. F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications*, 2003. Cambridge University Press. ISBN 0-521-78176-0. (Cited on pages 35 and 40.)
- H. Baumeister and D. Bert. Algebraic specification in CASL. In M. Frappier and H. Habrias, editors, *Software Specification Methods: An Overview Using a Case Study*, FACIT (Formal Approaches to Computing and Information Technology), chapter 12, pages 209–224. Springer, 2000. (Cited on page 156.)
- P. Baumgartner, A. Fuchs, and C. Tinelli. Darwin: A Theorem Prover for the Model Evolution Calculus. In S. Schulz, G. Sutcliffe, and T. Tammet, editors, *IJCAR Workshop on Empirically Successful First Order Reasoning (ESFOR (aka S4))*, Electronic Notes in Theoretical Computer Science, 2004. (Cited on page 191.)

- T. Berners-Lee, J. Hendler, and O. Lassila. The Semantic Web. *Scientific American*, May 2001. (Cited on page 27.)
- Y. Bertot and P. Castéran. *Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004. (Cited on page 81.)
- M. Bidoit and P. D. Mosses. *CASL User Manual*. LNCS Vol. 2900 (IFIP Series). Springer, 2004. (Cited on pages 27, 56, 62 and 142.)
- M. Bidoit, D. Sannella, and A. Tarlecki. Architectural specifications in CASL. *Formal Aspects of Computing*, 13:252–273, 2002a. (Cited on pages 15, 53 and 56.)
- M. Bidoit, D. Sannella, and A. Tarlecki. Global development via local observational construction steps. In K. Diks and W. Rytter, editors, *MFCS*, volume 2420 of *Lecture Notes in Computer Science*, pages 1–24. Springer, 2002b. ISBN 3-540-44040-2. (Cited on page 192.)
- M. Bidoit, D. Sannella, and A. Tarlecki. Observational interpretation of CASL specifications. *Mathematical Structures in Computer Science*, 18(2):325–371, 2008. (Cited on pages 16, 192 and 224.)
- P. Blackburn, M. de Rijke, and Y. Venema. *Modal Logic*. Cambridge University Press, Cambridge, England, 2001. ISBN 0 521 52714 7 (pbk). (Cited on page 31.)
- P. Blackburn, J. van Benthem, and F. Wolter. *Handbook of Modal Logic, Volume 3 (Studies in Logic and Practical Reasoning)*. Elsevier Science Inc., New York, NY, USA, 2006. ISBN 0444516905. (Cited on page 211.)
- M. Bortin, E. Broch Johnsen, and C. Lüth. Structured formal development in Isabelle. *Nordic Journal of Computing*, 12:1–20, 2006. (Cited on page 86.)
- T. Borzyszkowski. Moving specification structures between logical systems. In J. L. Fiadeiro, editor, *WADT 1998*, volume 1589 of *Lecture Notes in Computer Science*, pages 16–30. Springer, 1999. (Cited on page 35.)
- T. Borzyszkowski. Logical systems for structured specifications. *Theoretical Computer Science*, 286:197–245, 2002. (Cited on page 38.)
- A. Bouhoula, J.-P. Jouannaud, and J. Meseguer. Specification and proof in membership equational logic. *Theor. Comput. Sci.*, 236(1-2):35–132, 2000. (Cited on page 129.)
- P. Burmeister. Partial algebras — survey of a unifying approach towards a two-valued model theory for partial algebras. *Algebra Universalis*, 15:306–358, 1982. (Cited on page 35.)
- R. M. Burstall and J. A. Goguen. Putting theories together to make specifications. In *IJCAI*, pages 1045–1058, 1977. (Cited on page 17.)

- R.M. Burstall and J.A. Goguen. The semantics of CLEAR, a specification language. In *Proceedings of the Abstract Software Specifications, 1979 Copenhagen Winter School*, LNCS 86, pages 292–332. Springer, 1980. (Cited on pages 12 and 33.)
- L. Cheikhrouhou, G. Rock, W. Stephan, M. Schwan, and G. Lassmann. Verifying a chipcard-based biometric identification protocol in VSE. In J. Górski, editor, *SAFECOMP 2006*, volume 4166 of LNCS, pages 42–56. Springer, 2006. (Cited on page 197.)
- A. Church. A Formulation of the Simple Theory of Types. *Journal of Symbolic Logic*, 5(1):56–68, 1940. (Cited on page 80.)
- M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. The Maude 2.0 system. In R. Nieuwenhuis, editor, *Rewriting Techniques and Applications (RTA 2003)*, number 2706 in Lecture Notes in Computer Science, pages 76–87. Springer-Verlag, June 2003. (Cited on page 116.)
- M. Clavel, M. Palomino, and A. Riesco. Introducing the ITP tool: a tutorial. *Journal of Universal Computer Science*, 12(11):1618–1650, 2006. Programming and Languages. Special Issue with Extended Versions of Selected Papers from PROLE 2005: The Fifth Spanish Conference on Programming and Languages. (Cited on page 124.)
- M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, C. Talcott, A. Riesco, and A. Verdejo. Mobile Maude. In *All About Maude: A High-Performance Logical Framework*, volume 4350 of *Lecture Notes in Computer Science*, chapter 16, pages 485–522. Springer, 2007. (Cited on pages 17, 80, 123, 124 and 136.)
- M. Codescu. Generalized theoroidal institution comorphisms. In A. Corradini and U. Montanari, editors, *WADT 2008*, volume 5486 of *Lecture Notes in Computer Science*, pages 88–101. Springer, 2009. (Cited on page 106.)
- M. Codescu. Lambda expressions in CASL architectural specifications. In T. Mossakowski and H.-J. Kreowski, editors, *Recent Trends in Algebraic Development Techniques, 20th International Workshop, WADT 2010*, Lecture Notes in Computer Science. Springer, 2011. (Cited on page 158.)
- M. Codescu and T. Mossakowski. Heterogeneous colimits. In F. Boulanger, C. Gaston, and P.-Y. Schobbens, editors, *MoVaH'08 Workshop on Modeling, Validation and Heterogeneity*. IEEE press, 2008. (Cited on page 121.)
- M. Codescu and T. Mossakowski. Refinement trees: calculi, tools and applications. In A. Corradini and B. Klin, editors, *Algebra and Coalgebra in Computer Science, CALCO'11*, volume 6859 of *Lecture Notes in Computer Science*, pages 145–160. Springer, 2011. (Cited on page 192.)
- M. Codescu, F. Horozal, M. Kohlhase, T. Mossakowski, F. Rabe, and K. Sojakova. Towards logical frameworks in the heterogeneous tool set hets. In *Recent Trends*

- in Algebraic Development Techniques, 20th International Workshop, WADT 2010*, Lecture Notes in Computer Science. Springer, 2010a. (Cited on page 94.)
- M. Codescu, T. Mossakowski, A. Riesco, and C. Maeder. Integrating Maude into Hets. In Mike Johnson and Dusko Pavlovic, editors, *AMAST 2010*, volume 6486 of *Lecture Notes in Computer Science*, pages 60–75. Springer, 2010b. (Cited on page 136.)
- M. Codescu, B. Langenstein, C. Maeder, and T. Mossakowski. The VSE refinement method in HETS. *Electronic Communications of the EASST*, 2012. to appear. (Cited on page 220.)
- R. Constable, S. Allen, H. Bromley, W. Cleaveland, J. Cremer, R. Harper, D. Howe, T. Knoblock, N. Mendler, P. Panangaden, J. Sasaki, and S. Smith. *Implementing Mathematics with the Nuprl Development System*. Prentice-Hall, 1986. (Cited on page 81.)
- F. Cornelius, M. Baldamus, H. Ehrig, and F. Orejas. Abstract and behaviour module specifications. *Mathematical Structures in Computer Science*, 9(1):21–62, 1999. (Cited on page 101.)
- V. E. Căzănescu and G. Roşu. Weak inclusion systems. *Mathematical Structures in Computer Science*, 7(2):195–206, 1997. (Cited on page 37.)
- N. de Bruijn. The Mathematical Language AUTOMATH. In M. Laudet, editor, *Proceedings of the Symposium on Automated Demonstration*, volume 25 of *Lecture Notes in Mathematics*, pages 29–61. Springer, 1970. (Cited on page 79.)
- L. A. Dennis, G. Collins, M. Norrish, R. J. Boulton, K. Slind, and T. F. Melham. The PROSPER toolkit. *STTT*, 4(2):189–210, 2003. (Cited on page 196.)
- R. Diaconescu. Grothendieck institutions. *Applied categorical structures*, 10:383–402, 2002. (Cited on pages 13, 17, 43, 107, 110 and 121.)
- R. Diaconescu. *Institution-independent Model Theory*. Birkhäuser Basel, 2008. ISBN 3764387076, 9783764387075. (Cited on pages 124, 126 and 211.)
- R. Diaconescu, J.A. Goguen, and P. Stefaneas. Logical Support for Modularisation. In *2nd Workshop on Logical Environments*, pages 83–130. CUP, New York, 1993. (Cited on pages 17 and 38.)
- D. Dietrich, L. Schröder, and E. Schulz. Formalizing and operationalizing industrial standards. In D. Giannakopoulou and F. Orejas, editors, *Fundamental Approaches to Software Engineering*, volume 6603 of *Lecture Notes in Computer Science*, pages 81–95. Springer Berlin / Heidelberg, 2011. (Cited on page 29.)
- L. Dixon and J. D. Fleuriot. Higher order rippling in IsaPlanner. In *TPHOLs*, pages 83–98, 2004. (Cited on pages 124 and 136.)

- F. Durán and J. Meseguer. A Church-Rosser checker tool for conditional order-sorted equational Maude specifications. In *Proceedings of the 8th International Workshop on Rewriting Logic and its Applications (WRLA 2010)*, Lecture Notes in Computer Science, 2010. To appear. (Cited on pages 116 and 136.)
- F. Durán, S. Lucas, and J. Meseguer. MTT: The Maude Termination Tool (system description). In A. Armando, P. Baumgartner, and G. Dowek, editors, *Proceedings of the 4th International Joint Conference on Automated Reasoning, IJCAR 2008, Sydney, Australia, August 12-15*, volume 5195 of *Lecture Notes in Computer Science*, pages 313–319. Springer, 2008. (Cited on page 136.)
- H. Ehrig, H.-J. Kreowski, B. Mahr, and P. Padawitz. Algebraic implementation of abstract data types. *Theor. Comput. Sci.*, 20:209–263, 1982. (Cited on page 14.)
- H. Ehrig, P. Pepper, and F. Orejas. On recent trends in algebraic specification. In *Proc. ICALP'89*, volume 372 of *Lecture Notes in Computer Science*, pages 263–288. Springer Verlag, 1989. (Cited on page 101.)
- J.S. Fitzgerald and C. B. Jones. Modularizing the formal description of a database system. In D. Bjørner, C. A. R. Hoare, and H. Langmaack, editors, *VDM Europe*, volume 428 of *Lecture Notes in Computer Science*, pages 189–210. Springer, 1990. ISBN 3-540-52513-0. (Cited on page 15.)
- K. Futatsugi and R. Diaconescu. *CafeOBJ Report*. World Scientific, AMAST Series, 1998. (Cited on pages 13, 125 and 126.)
- C. Gârlea. An Extended Modal Logic Institution, 2011. Master's thesis, Școala Normală Superioară Bucharest, Romania. (Cited on page 27.)
- V. Giarratana, F. Gimona, and U. Montanari. Observability concepts in abstract data type specifications. In A. W. Mazurkiewicz, editor, *MFCS*, volume 45 of *Lecture Notes in Computer Science*, pages 576–587. Springer, 1976. (Cited on page 15.)
- J. Goguen and G. Roșu. Institution morphisms. *Formal Aspects of Computing*, 13: 274–307, 2002. (Cited on pages 13, 40, 41, 96 and 127.)
- J. A. Goguen and R. M. Burstall. Institutions: Abstract model theory for specification and programming. *Journal of the Association for Computing Machinery*, 39:95–146, 1992. (Cited on pages 12, 33, 35 and 38.)
- J. A. Goguen and J. Meseguer. Universal realization, persistent interconnection and implementation of abstract modules. In M. Nielsen and E. M. Schmidt, editors, *ICALP*, volume 140 of *Lecture Notes in Computer Science*, pages 265–281. Springer, 1982. ISBN 3-540-11576-5. (Cited on page 14.)
- S. Gröning. Beweisunterstützung für HasCASL in Isabelle /HOL. Master's thesis, University of Bremen, 2005. Diplomarbeit. (Cited on page 30.)

- E. H. Haeusler, A. Martini, and U. Wolter. Some models of heterogeneous and distributed specifications based on universal constructions. In J.-Y. Béziau and A. Costa-Leite, editors, *Perspectives on Universal Logic*, pages 297–318. Polimetrica Publisher, Italy, 2007. (Cited on page 121.)
- R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, 1993. (Cited on pages 79 and 80.)
- R. Harper, D. Sannella, and A. Tarlecki. Structured presentations and logic representations. *Annals of Pure and Applied Logic*, 67:113–160, 1994. (Cited on pages 79, 80 and 81.)
- M. Herbstritt. zChaff: Modifications and extensions. report00188, Institut für Informatik, Universität Freiburg, July 17 2003. (Cited on page 28.)
- C. A. R. Hoare. Proof of correctness of data representations. *Acta Inf.*, 1:271–281, 1972. (Cited on pages 61 and 205.)
- P. Hoffman. Verifying generative CASL architectural specifications. In M. Wirsing, D. Pattinson, and R. Hennicker, editors, *Recent Trends in Algebraic Development Techniques, 16th International Workshop, WADT 2002, Frauenchiemsee, Germany, 2002, Revised Selected Papers*, LNCS Vol. 2755, pages 233–252. Springer, 2003. (Cited on page 154.)
- P. Hoffman. *Architectural Specifications and Their Verification*. PhD thesis, Warsaw University, 2005. (Cited on pages 154 and 166.)
- F. Horozal. Logic translations with declaration patterns. <https://svn.kwarc.info/repos/fhorozal/pubs/patterns.pdf>, 2012. (Cited on page 89.)
- F. Horozal and F. Rabe. Representing Model Theory in a Type-Theoretical Logical Framework. *Theoretical Computer Science*, 412(37):4919–4945, 2011. (Cited on pages 83 and 92.)
- D. Hutter, B. Langenstein, G. Rock, Jörg Siekmann, W. Stephan, and R. Vogt. Formal software development in the verification support environment. *Journal of Experimental and Theoretical Artificial Intelligence*, 12(4):383–406, December 2000. (Cited on page 197.)
- M. Iancu and F. Rabe. Formalizing Foundations of Mathematics. *Mathematical Structures in Computer Science*, 21(4):883–911, 2011. (Cited on page 82.)
- M. Kohlhase, T. Mossakowski, and F. Rabe. The LATIN Project, 2009. see <https://trac.omdoc.org/LATIN/>. (Cited on pages 80 and 82.)
- O. Kutz and T. Mossakowski. A modular consistency proof for Dolce. In *25th conference on Artificial Intelligence, AAI-11*, 2011. To appear. (Cited on page 188.)

- B. Langenstein, R. Vogt, and Markus Ullmann. The use of formal methods for trusted digital signature devices. In J. N. Etheredge and B. Z. Manaris, editors, *FLAIRS Conference*, pages 336–340. AAAI Press, 2000. ISBN 1-57735-113-4. (Cited on page 197.)
- M. Leuschel and H. Wehrheim, editors. *Integrated Formal Methods, 7th International Conference, IFM 2009, Düsseldorf, Germany, February 16-19, 2009. Proceedings*, volume 5423 of *Lecture Notes in Computer Science*, 2009. Springer. ISBN 978-3-642-00254-0. (Cited on page 196.)
- M. Liu. Konsistenz-Check von CASL-Spezifikationen. Master’s thesis, University of Bremen, 2008. (Cited on pages 189 and 191.)
- C. Lüth, M. Roggenbach, and L. Schröder. CCC - the CASL consistency checker. In J.L. Fiadeiro, editor, *Recent Trends in Algebraic Development Techniques, 17th International Workshop (WADT 2004)*, volume 3423 of *Lecture Notes in Computer Science*, pages 94–105. Springer; Berlin; <http://www.springer.de>, 2005. (Cited on page 106.)
- K. Lüttich and T. Mossakowski. Reasoning Support for CASL with Automated Theorem Proving Systems. In J. Fiadeiro, editor, *WADT 2006*, LNCS 4409, pages 74–91. Springer-Verlag, 2007. (Cited on pages 28, 30, 98 and 99.)
- K. Lüttich, T. Mossakowski, and B. Krieg-Brückner. Ontologies for the Semantic Web in CASL. In *17th Int. Workshop on Algebraic Development Techniques (WADT 2004)*, volume 3423 of *LNCS*, pages 106–125. Springer, 2005. (Cited on page 28.)
- C. Lutz, Dirk Walther, and F. Wolter. Conservative Extensions in Expressive Description Logics. In Manuela M. Veloso, editor, *Proceedings of IJCAI 2007: the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6–12, 2007*, pages 453–458. AAAI Press, 2007. (Cited on page 38.)
- S. Mac Lane. *Categories for the Working Mathematician*. Springer, 1971. (Cited on pages 33, 111 and 115.)
- N. Martí-Oliet and J. Meseguer. General logics and logical frameworks. In Dov M. Gabbay, editor, *What is a logical system?*, pages 355–391. Oxford University Press, Inc., New York, NY, USA, 1994. ISBN 0-19-853859-6. (Cited on pages 17, 80 and 86.)
- N. Martí-Oliet, J. Meseguer, and M. Palomino. Theoroidal maps as algebraic simulations. In J. L. Fiadeiro, P. D. Mosses, and F. Orejas, editors, *WADT 2004*, volume 3423 of *LNCS*, pages 126–143. Springer, 2004. (Cited on page 110.)
- P. Martin-Löf. An Intuitionistic Theory of Types: Predicative Part. In *Proceedings of the '73 Logic Colloquium*, pages 73–118. North-Holland, 1974. (Cited on page 81.)

- J. Meng, C. Quigley, and L. Paulson. Automation for interactive proof: First prototype. *Inf. Comput.*, 204(10):1575–1596, 2006. (Cited on page 196.)
- J. Meseguer. General logics. In H.-D. Ebbinghaus et al., editors, *Proceedings, Logic Colloquium, 1987*, pages 275–329. North-Holland, 1989. (Cited on pages 17, 39, 96 and 104.)
- J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992. (Cited on pages 80, 124 and 125.)
- J. Meseguer. Membership algebra as a logical framework for equational specification. In F. Parisi-Presicce, editor, *Recent Trends in Algebraic Development Techniques, 12th International Workshop, WADT'97, Tarquinia, Italy, June 3–7, 1997, Selected Papers*, volume 1376 of *Lecture Notes in Computer Science*, pages 18–61. Springer, 1998. (Cited on page 126.)
- Robin Milner. An algebraic definition of simulation between programs. In *IJCAI*, pages 481–489, 1971. (Cited on page 61.)
- T. Mossakowski. *Representations, hierarchies and graphs of institutions*. PhD thesis, Universität Bremen, 1996. Also appeared as book in Logos Verlag. (Cited on pages 95 and 102.)
- T. Mossakowski. Colimits of order-sorted specifications. In F. Parisi Presicce, editor, *Recent trends in algebraic development techniques. Proc. 12th International Workshop*, volume 1376 of *Lecture Notes in Computer Science*, pages 316–332. Springer Verlag, London, 1998. URL http://dx.doi.org/10.1007/3-540-64299-4_42. (Cited on page 116.)
- T. Mossakowski. Specification in an arbitrary institution with symbols. In C. Choppy, D. Bert, and P. Mosses, editors, *Recent Trends in Algebraic Development Techniques, 14th International Workshop, WADT'99, Bonas, France*, volume 1827 of *Lecture Notes in Computer Science*, pages 252–270. Springer Verlag, London, 2000. ISBN 3-540-67898-0. URL [http://www.springerlink.com/\(bt4qw245oavupgzdxw3zpuul\)/app/home/contribution.asp?referrer=parent&backto=searchcitationsresults,8,38;](http://www.springerlink.com/(bt4qw245oavupgzdxw3zpuul)/app/home/contribution.asp?referrer=parent&backto=searchcitationsresults,8,38;). (Cited on pages 37 and 57.)
- T. Mossakowski. Comorphism-based Grothendieck logics. In K. Diks and W. Rytter, editors, *Mathematical foundations of computer science*, volume 2420 of *Lecture Notes in Computer Science*, pages 593–604. Springer Verlag, London, 2002a. (Cited on pages 14, 43, 59, 106 and 107.)
- T. Mossakowski. Relating CASL with other specification languages: the institution level. *Theoretical Computer Science*, 286:367–475, 2002b. (Cited on pages 29, 35, 41, 97, 99 and 211.)

- T. Mossakowski. Foundations of heterogeneous specification. In M. Wirsing, D. Pattinson, and R. Hennicker, editors, *Recent Trends in Algebraic Development Techniques, 16th International Workshop, WADT 2002, Frauenchiemsee, Germany, 2002, Revised Selected Papers*, Lecture Notes in Computer Science, pages 359–375. Springer Verlag, London, 2003. ISBN 3-540-20537-3. (Cited on pages 14, 42 and 95.)
- T. Mossakowski. ModalCASL - specification with multi-modal logics. language summary, 2004. (Cited on page 27.)
- T. Mossakowski. Heterogeneous specification and the heterogeneous tool set. Technical report, Universitaet Bremen, 2005. Habilitation thesis. (Cited on pages 13, 14, 17, 23, 59, 99, 110, 118, 121 and 136.)
- T. Mossakowski. Institutional 2-cells and Grothendieck institutions. In K. Futatsugi, J.-P. Jouannaud, and J. Meseguer, editors, *Algebra, Meaning and Computation. Essays Dedicated to Joseph A. Goguen on the Occasion of His 65th Birthday*, volume 4060 of *LNCS*, pages 124–149. Springer; Berlin, 2006. (Cited on page 14.)
- T. Mossakowski and A. Tarlecki. Heterogeneous logical environments for distributed specifications. In A. Corradini and U. Montanari, editors, *WADT 2008*, volume 5486 of *Lecture Notes in Computer Science*, pages 266–289. Springer, 2009. (Cited on page 44.)
- T. Mossakowski, S. Autexier, and D. Hutter. Extending development graphs with hiding. In *Fundamental Approaches to Software Engineering*, volume 2029 of *Lecture Notes in Computer Science*, pages 269–283. Springer Verlag, London, 2001. (Cited on pages 53 and 119.)
- T. Mossakowski, D. Sannella, and A. Tarlecki. A simple refinement language for CASL. In J. L. Fiadeiro, editor, *WADT 2004*, volume 3423 of *Lecture Notes in Computer Science*, pages 162–185. Springer Verlag, 2005. (Cited on pages 15, 61, 66, 68, 69, 71, 73, 74 and 192.)
- T. Mossakowski, S. Autexier, and D. Hutter. Development graphs—proof management for structured specifications. *Journal of Logic and Algebraic Programming*, 67(1–2):114–145, 2006a. (Cited on pages 51, 108 and 113.)
- T. Mossakowski, L. Schröder, M. Roggenbach, and H. Reichel. Algebraic-coalgebraic specification in CoCASL. *Journal of Logic and Algebraic Programming*, 67(1–2):146–197, 2006b. URL <http://dx.doi.org/10.1016/j.jlap.2005.09.006>. Extends (Mossakowski et al. 2003). (Cited on page 27.)
- T. Mossakowski, A. Haxthausen, D. Sannella, and A. Tarlecki. CASL — The Common Algebraic Specification Language. In D. Bjørner and M. Henson, editors, *Logics of Specification Languages*, pages 241–298. Springer, 2008. (Cited on page 46.)

- T. Mossakowski, C. Maeder, and M. Codescu. HETS User Guide. Manuscript, 2011. (Cited on page 23.)
- P. D. Mosses. *Casl Reference Manual: The Complete Documentation Of The Common Algebraic Specification Language (LECTURE NOTES IN COMPUTER SCIENCE)*. SpringerVerlag, 2004. ISBN 3540213015. (Cited on pages 12, 15, 27, 29, 37, 46, 49, 50, 56, 57, 58, 59, 69, 73, 139, 140, 142, 143, 146, 148, 154, 158, 159, 160, 161, 163, 192 and 211.)
- B. Motik, B. Cuenca Grau, I. Horrocks, Z. Wu, A. Fokoue, and C. Lutz. OWL 2 web ontology language: Profiles. W3C recommendation, World Wide Web Consortium (W3C), October 2009a. URL <http://www.w3.org/TR/2009/REC-owl2-profiles-20091027/>. (Cited on page 28.)
- B. Motik, P. F. Patel-Schneider, and B. Cuenca Grau. OWL 2 web ontology language: Direct semantics. W3C recommendation, World Wide Web Consortium (W3C), October 2009b. URL <http://www.w3.org/TR/2009/REC-owl2-direct-semantics-20091027/>. (Cited on page 28.)
- T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. Springer Verlag, 2002. (Cited on pages 28, 80, 135, 136 and 195.)
- U. Norell. The Agda Wiki, 2005. <http://wiki.portal.chalmers.se/agda>. (Cited on page 81.)
- L. Paulson. *Isabelle: a generic theorem prover*, volume 828 of *Lecture Notes in Computer Science*. Springer, 1994. (Cited on pages 79 and 80.)
- L. Paulson and M. Coen. Zermelo-Fraenkel Set Theory, 1993. Isabelle distribution, ZF/ZF.thy. (Cited on page 81.)
- B. Pelzer and C. Wernhard. System description: E-krhyper. In Frank Pfenning, editor, *CADE*, volume 4603 of *Lecture Notes in Computer Science*, pages 508–513. Springer, 2007. ISBN 978-3-540-73594-6. (Cited on page 28.)
- S. Peyton-Jones, editor. *Haskell 98 Language and Libraries — The Revised Report*. Cambridge, 2003. also: *J. Funct. Programming* **13** (2003). (Cited on pages 24 and 27.)
- F. Pfenning. Structural cut elimination: I. intuitionistic and classical logic. *Information and Computation*, 157(1-2):84–141, 2000. (Cited on page 81.)
- F. Pfenning and C. Schürmann. System description: Twelf - a meta-logical framework for deductive systems. *Lecture Notes in Computer Science*, 1632:202–206, 1999. (Cited on page 80.)
- F. Pfenning, C. Schürmann, M. Kohlhase, N. Shankar, and S. Owre. The Logosphere Project, 2003. <http://www.logosphere.org/>. (Cited on page 80.)

- A. Poswolsky and C. Schürmann. System Description: Delphin - A Functional Programming Language for Deductive Systems. In A. Abel and C. Urban, editors, *International Workshop on Logical Frameworks and Metalanguages: Theory and Practice*, pages 135–141. ENTCS, 2008. (Cited on page 81.)
- F. Rabe. First-Order Logic with Dependent Types. In N. Shankar and U. Furbach, editors, *Proceedings of the 3rd International Joint Conference on Automated Reasoning*, volume 4130 of *Lecture Notes in Computer Science*, pages 377–391. Springer, 2006. (Cited on page 29.)
- F. Rabe. *Representing Logics and Logic Translations*. PhD thesis, Jacobs University Bremen, 2008. see <http://kwarc.info/frabe/Research/phdthesis.pdf>. (Cited on pages 16, 92 and 94.)
- F. Rabe. A Logical Framework Combining Model and Proof Theory. see http://kwarc.info/frabe/Research/rabe_combining_09.pdf, 2010. (Cited on pages 84, 89 and 90.)
- F. Rabe and C. Schürmann. A Practical Module System for LF. In J. Cheney and A. Felty, editors, *Proceedings of the Workshop on Logical Frameworks: Meta-Theory and Practice (LFMTP)*, volume LFMTP'09 of *ACM International Conference Proceeding Series*, pages 40–48. ACM Press, 2009. (Cited on pages 81 and 82.)
- H. Reichel. *Initial computability, algebraic specifications, and partial algebras*. Oxford University Press, Inc., New York, NY, USA, 1987. ISBN 0-198-53806-5. (Cited on page 132.)
- W. Reif. Verification of large software systems. In R. K. Shyamasundar, editor, *FSTTCS*, volume 652 of *LNCS*, pages 241–252. Springer, 1992. ISBN 3-540-56287-7. (Cited on pages 205 and 213.)
- A. Riazanov and A. Voronkov. The design and implementation of VAMPIRE. *AI Communications*, 15(2-3):91–110, 2002. (Cited on page 28.)
- M. Roggenbach. CSP-CASL - a new integration of process algebra and algebraic specification. *Theoretical Computer Science*, 354(1):42–71, 2006. (Cited on page 27.)
- M. Roggenbach and L. Schröder. Towards trustworthy specifications I: Consistency checks. In M. Cerioli and G. Reggio, editors, *Recent Trends in Algebraic Specification Techniques, 15th International Workshop, WADT 2001*, volume 2267 of *Lecture Notes in Computer Science*. Springer; Berlin; <http://www.springer.de>, 2001. (Cited on page 189.)
- D. Sannella and A. Tarlecki. On observational equivalence and algebraic specification. *J. Comput. Syst. Sci.*, 34(2/3):150–178, 1987. (Cited on page 15.)

- D. Sannella and A. Tarlecki. Toward formal development of programs from algebraic specifications: implementations revisited. *Acta Informatica*, 25:233–281, 1988a. (Cited on pages 14, 72, 139, 189 and 192.)
- D. Sannella and A. Tarlecki. Specifications in an arbitrary institution. *Inform. and Comput.*, 76:165–210, 1988b. (Cited on pages 12, 38 and 47.)
- D. Sannella and A. Tarlecki. Mind the gap! Abstract versus concrete models of specifications. In *Proc. 21st Intl. Symp. on Mathematical Foundations of Computer Science*, volume 1113 of *Lecture Notes in Computer Science*, pages 114–134. Springer, 1996. (Cited on page 15.)
- D. Sannella and A. Tarlecki. Essential concepts of algebraic specification and program development. *Formal Asp. Comput.*, 9(3):229–269, 1997. (Cited on page 15.)
- D. Sannella and A. Tarlecki. *Foundations of Algebraic Specification and Formal Software Development*. EATCS Monographs in Theoretical Computer Science. Springer, 2012. ISBN 978-3-642-17335-6. (Cited on pages 16, 29, 50, 72, 206 and 211.)
- M. Schneider. OWL 2 web ontology language: RDF-based semantics. W3C recommendation, World Wide Web Consortium (W3C), October 2009. URL <http://www.w3.org/TR/2009/REC-owl2-rdf-based-semantics-20091027/>. (Cited on page 28.)
- W. M. Schorlemmer and Y. Kalfoglou. Institutionalising ontology-based semantic integration. *Applied Ontology*, 3(3):131–150, 2008. (Cited on pages 28, 35, 109, 118 and 120.)
- L. Schröder. Higher order and reactive algebraic specification and development. Summary of papers constituting a cumulative habilitation thesis; available under <http://www.informatik.uni-bremen.de/~lschrode/papers/Summary.pdf>, 2005. (Cited on page 27.)
- L. Schröder. The HasCASL prologue - categorical syntax and semantics of the partial λ -calculus. *Theoret. Comput. Sci.*, 353:1–25, 2006. (Cited on page 27.)
- L. Schröder and T. Mossakowski. HasCASL: towards integrated specification and development of functional programs. In H. Kirchner and C. Ringeissen, editors, *Algebraic Methodology And Software Technology (AMAST 2002)*, volume 2422 of *Lecture Notes in Computer Science*, pages 99–116, 2002. (Cited on page 27.)
- L. Schröder, T. Mossakowski, P. Hoffman, B. Klin, and A. Tarlecki. Semantics of architectural specifications in CASL. In *Fundamental Approaches to Software Engineering*, volume 2029 of *LNCS*, pages 253–268. Springer, 2001. (Cited on page 58.)

- L. Schröder, T. Mossakowski, and C. Maeder. HASCASL – Integrated functional specification and programming. Language summary. Available at http://www.informatik.uni-bremen.de/agbkb/forschung/formal_methods/CoFI/HasCASL, 2003. (Cited on page 27.)
- L. Schröder, T. Mossakowski, A. Tarlecki, P. Hoffman, and B. Klin. Amalgamation in the semantics of CASL. *Theoretical Computer Science*, 331(1):215–247, 2005. ISSN 0304-3975. (Cited on pages 38 and 110.)
- S. Schulz. E – A Brainiac Theorem Prover. *Journal of AI Communications*, 15(2/3): 111–126, 2002. (Cited on page 28.)
- C. Schürmann and M. Stehr. An Executable Formalization of the HOL/Nuprl Connection in the Metalogical Framework Twelf. In *11th International Conference on Logic for Programming Artificial Intelligence and Reasoning*, 2004. (Cited on page 81.)
- D. R. Smith. Designware: Software development by refinement. *Electr. Notes Theor. Comput. Sci.*, 29:275–287, 1999. (Cited on page 61.)
- K. Sojakova. Mechanically Verifying Logic Translations, 2010. Master’s thesis, Jacobs University Bremen. (Cited on page 89.)
- G. Sutcliffe. The TPTP world – infrastructure for automated reasoning. In E. M. Clarke and A. Voronkov, editors, *LPAR (Dakar)*, volume 6355 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 2010. ISBN 978-3-642-17510-7. (Cited on page 28.)
- A. Tarlecki. Moving between logical systems. In M. Haverdaen, O. Owe, and O.-J. Dahl, editors, *WADT 1995*, LNCS 1130, pages 478–502. Springer Verlag, 1996. (Cited on pages 36, 42 and 79.)
- A. Tarlecki. Towards heterogeneous specifications. In *In Frontiers of Combining Systems FroCoS’98, Applied Logic Series*, pages 337–360. Kluwer Academic Publishers, 1998. (Cited on page 50.)
- P. Torrini, C. Lüth, C. Maeder, and T. Mossakowski. Translating Haskell to Isabelle. Isabelle workshop at CADE-21, 2007. (Cited on page 30.)
- A. Trybulec and H. Blair. Computer Assisted Reasoning with MIZAR. In A. Joshi, editor, *Proceedings of the 9th International Joint Conference on Artificial Intelligence*, pages 26–28, 1985. (Cited on page 81.)
- T. Weber. Bounded model generation for Isabelle/HOL. volume 125 of *Electronic Notes in Theoretical Computer Science*, pages 103–116, 2005. (Cited on page 191.)
- C. Weidenbach, U. Brahm, T. Hillenbrand, E. Keen, C. Theobalt, and D. Topic. SPASS version 2.0. In A. Voronkov, editor, *Automated Deduction, CADE-18*, volume 2392

- of *LNCS Vol.* , pages 275–279. Springer, 2002. ISBN 3-540-43931-5. (Cited on pages 28, 191 and 195.)
- K. E. Williamson, M. Healy, and R. A. Barker. Industrial applications of software synthesis via category theory-case studies using Specware. *Autom. Softw. Eng.*, 8(1):7–30, 2001. (Cited on page 107.)
- M. Wirsing. Algebraic specification languages: An overview. In E. Astesiano, G. Reggio, and A. Tarlecki, editors, *Recent Trends in Data Type Specification*, volume 906 of *Lecture Notes in Computer Science*, pages 81–115. Springer Berlin / Heidelberg, 1995. ISBN 978-3-540-59132-0. (Cited on page 12.)
- N. Wirth. Program development by stepwise refinement. *Commun. ACM*, 14(4):221–227, 1971. (Cited on page 15.)
- S. Wölfl, T. Mossakowski, and L. Schröder. Qualitative constraint calculi: Heterogeneous verification of composition tables. In *Proceedings of the Twentieth International Florida Artificial Intelligence Research Society Conference (FLAIRS 2007)*, pages 665–670. AAAI Press, 2007. (Cited on page 14.)
- J. Zimmer and S. Autexier. The MathServe System for Semantic Web Reasoning Services. In U. Furbach and N. Shankar, editors, *3rd IJCAR*, LNCS 4130. Springer, 2006. (Cited on page 28.)
- A. Zimmermann, M. Krötzsch, J. Euzenat, and P. Hitzler. Formalizing Ontology Alignment and its Operations with Category Theory. In B. Bennett and C. Fellbaum, editors, *FOIS 2006*, volume 150 of *Frontiers in Artificial Intelligence and Applications*, pages 277–288. IOS Press, NOV 2006. (Cited on page 118.)

Part V

Appendices

Natural numbers as binary words

```

spec NATS_SIG =
  sort   nats
  op    zero_n : nats
  op    succ_n : nats → nats
  op    prdc_n : nats → nats
  op    add_n  : nats × nats → nats

spec SIMPNATS =
  NATS_SIG
then free type nats ::= zero_n | succ_n(nats)
  vars   m, n : nats
  • prdc_n(zero_n) = zero_n
  • prdc_n(succ_n(m)) = m
  • add_n(m, zero_n) = m
  • add_n(m, succ_n(n)) = succ_n(add_n(m, n))

```

logic VSE

```

spec SIMPNATS_GOALS =
  SIMPNATS with logic → CASL2VSERefine

```

logic CASL

```

spec BIN_DATA =
  free type bin ::= b_zero | b_one | s0(bin) | s1(bin)
  op    pop : bin → bin
  var   x : bin
  • pop(s0(x)) = x
  • pop(s1(x)) = x

```

```

spec BIN =
  BIN_DATA
then op   top : bin → bin
  vars   x, y, z : bin
  • top(b_zero) = b_zero
  • top(b_one) = b_one

```

- $top(s0(x)) = b_zero$
- $top(s1(x)) = b_one$

logic VSE

```

spec NATS_IMPL =
  BIN with logic → CASL2VSEImport
then PROCEDURES
  hnz : IN bin;
  nlz : IN bin;
  i_badd : IN bin, IN bin, OUT bin, OUT bin;
  i_add : IN bin, IN bin → bin;
  i_prdc : IN bin → bin;
  i_succ : IN bin → bin;
  i_zero : → bin;
  eq : IN bin, IN bin → Boolean
  • DEFPROCS
    PROCEDURE hnz(x)
    BEGIN
    IF x = b_zero
    THEN ABORT
    ELSE IF x = b_one THEN SKIP ELSE hnz(pop(x)) FI
    FI
    END;

    PROCEDURE nlz(x)
    BEGIN IF x = b_zero THEN SKIP ELSE hnz(x) FI END
    DEFPROCSEND
  • DEFPROCS
    PROCEDURE i_badd(a, b, z, c)
    BEGIN
    IF a = b_zero
    THEN c := b_zero;
      z := b
    ELSE c := b;
      IF b = b_one
      THEN z := b_zero
      ELSE z := b_one
      FI
    FI
    END;

  FUNCTION i_add(x, y)

```

%(restr)%

```

BEGIN
DECLARE
z : bin := b_zero, c : bin := b_zero, s : bin := b_zero;
IF x = b_zero
THEN s := y
ELSE IF y = b_zero
    THEN s := x
    ELSE IF x = b_one
        THEN s := i_succ(y)
        ELSE IF y = b_one
            THEN s := i_succ(x)
            ELSE i_badd(top(x), top(y), z, c);
                IF c = b_one
                    THEN s := i_add(pop(x), pop(y))
                    ELSE s := i_succ(pop(x));
                        s := i_add(s, pop(y))
                FI;
                IF z = b_zero
                    THEN s := s0(s)
                    ELSE s := s1(s)
                FI
            FI
        FI
    FI
FI;
RETURN s
END;

FUNCTION i_prdc(x)
BEGIN
DECLARE
y : bin := b_zero;
IF x = b_zero  $\vee$  x = b_one
THEN y := b_zero
ELSE IF x = s0(b_one)
    THEN y := b_one
    ELSE IF top(x) = b_one
        THEN y := s0(pop(x))
        ELSE y := i_prdc(pop(x));
            y := s1(y)
        FI
    FI
FI;
RETURN y

```

END;

```

FUNCTION i_succ(x)
BEGIN
DECLARE
y : bin := b_one;
IF x = b_zero
THEN y := b_one
ELSE IF x = b_one
THEN y := s0(b_one)
ELSE IF top(x) = b_zero
THEN y := s1(pop(x))
ELSE y := i_succ(pop(x));
y := s0(y)
FI
FI
RETURN y
END;
```

```

FUNCTION i_zero()
BEGIN RETURN b_zero END
DEFPROCSEND
```

%(impl)%

- DEFPROCS

```

FUNCTION eq(x, y)
BEGIN
DECLARE
res : Boolean := False;
IF x = y THEN res := True FI;
RETURN res
END
DEFPROCSEND
```

%(congruence)%

logic VSE

refinement BINARY_ARITH =
NATS **refined via**
logic \rightarrow CASL2VSERefine,
nats \mapsto bin, gn_restr_nats \mapsto nlz, gn_eq_nats \mapsto eq,
gn_zero_n \mapsto i_zero, gn_succ_n \mapsto i_succ,
gn_prdc_n \mapsto i_prdc, gn_add_n \mapsto i_add
to NATS_IMPL

Steam boiler control system

library USERMANUAL/SBCS

%author Michel Bidoit <bidoit@lsv.ens-cachan.fr>

%date 20 Oct 2003

%display(*__half* %LATEX *__/2*)%

%display(*__square* %LATEX *__^2*)%

from BASIC/NUMBERS **get** NAT

from BASIC/STRUCTURED DATATYPES **get** SET

from BASIC/STRUCTURED DATATYPES **get** TOTALMAP

spec VALUE =

%% At this level we don't care about the exact specification of values.

NAT

then sort *Nat < Value*

ops *__+__* : *Value* × *Value* → *Value*, *assoc*, *comm*, *unit 0*;

__-__ : *Value* × *Value* → *Value*;

*__*__* : *Value* × *Value* → *Value*, *assoc*, *comm*, *unit 1*;

__/2, *__^2* : *Value* → *Value*;

min, *max* : *Value* × *Value* → *Value*

preds *__<__*, *__<=__* : *Value* × *Value*

spec BASICS =

free type

PumpNumber ::= *Pump1* | *Pump2* | *Pump3* | *Pump4*

free type *PumpState* ::= *Open* | *Closed*

free type *PumpControllerState* ::= *Flow* | *NoFlow*

free type

PhysicalUnit

::= *Pump(PumpNumber)*

| *PumpController(PumpNumber)*

| *SteamOutput*

| *WaterLevel*

free type

Mode

::= *Initialization*

| *Normal*

```

| Degraded
| Rescue
| EmergencyStop

```

```

spec MESSAGES_SENT =
  BASICS

```

```

then free type

```

```

  S_Message
  ::= MODE(Mode)
  | PROGRAM_READY
  | VALVE
  | OPEN_PUMP(PumpNumber)
  | CLOSE_PUMP(PumpNumber)
  | FAILURE_DETECTION(PhysicalUnit)
  | REPAIRED_ACKNOWLEDGEMENT(PhysicalUnit)

```

```

spec MESSAGES_RECEIVED =
  BASICS and VALUE

```

```

then free type

```

```

  R_Message
  ::= STOP
  | STEAM_BOILER_WAITING
  | PHYSICAL_UNITS_READY
  | PUMP_STATE(PumpNumber; PumpState)
  | PUMP_CONTROLLER_STATE(PumpNumber;
                           PumpControllerState)

  | LEVEL(Value)
  | STEAM(Value)
  | REPAIRED(PhysicalUnit)
  | FAILURE_ACKNOWLEDGEMENT(PhysicalUnit)
  | junk

```

```

spec SBCS_CONSTANTS =
  VALUE

```

```

then ops C, M1, M2, N1, N2, W, U1, U2, P : Value;

```

```

  dt : Value

```

```

  %% Time duration between two cycles (5 sec.)

```

```

  %% These constants must verify some obvious properties:

```

- $0 < M1$
- $M1 < N1$
- $N1 < N2$
- $N2 < M2$
- $M2 < C$
- $0 < W$

- $0 < U1$
- $0 < U2$
- $0 < P$

```

spec PRELIMINARY =
  SET[MESSAGES_RECEIVED fit Elem  $\mapsto$  R_Message]
and SET[MESSAGES_SENT fit Elem  $\mapsto$  S_Message]
and SBCS_CONSTANTS

spec SBCS_STATE_1 =
  PRELIMINARY
then sort State
  ops mode : State  $\rightarrow$  Mode;
        numSTOP : State  $\rightarrow$  Nat

spec MODE_EVOLUTION
  [preds Transmission_OK : State  $\times$  Set[R_Message];
        PU_OK : State  $\times$  Set[R_Message]  $\times$  PhysicalUnit;
        DangerousWaterLevel : State  $\times$  Set[R_Message]]
  given SBCS_STATE_1 =
  local

  %% Auxiliary predicates to structure the specification of next_mode.
  preds Everything_OK, AskedToStop, SystemStillControllable,
        Emergency : State  $\times$  Set[R_Message]
   $\forall s : \textit{State}; \textit{msgs} : \textit{Set}[\textit{R\_Message}]$ 
  • Everything_OK(s, msgs)
     $\Leftrightarrow$  Transmission_OK(s, msgs)
     $\wedge \forall pu : \textit{PhysicalUnit}$  • PU_OK(s, msgs, pu)
  • AskedToStop(s, msgs)  $\Leftrightarrow$  numSTOP(s) = 2  $\wedge$  STOP eps msgs
  • SystemStillControllable(s, msgs)
     $\Leftrightarrow$  PU_OK(s, msgs, SteamOutput)
     $\wedge \exists pn : \textit{PumpNumber}$ 
      • PU_OK(s, msgs, Pump(pn))
       $\wedge$  PU_OK(s, msgs, PumpController(pn))
  • Emergency(s, msgs)
     $\Leftrightarrow$  mode(s) = EmergencyStop  $\vee$  AskedToStop(s, msgs)
     $\vee \neg$  Transmission_OK(s, msgs)
     $\vee$  DangerousWaterLevel(s, msgs)
     $\vee (\neg \textit{PU\_OK}(s, \textit{msgs}, \textit{WaterLevel})$ 
       $\wedge \neg \textit{SystemStillControllable}(s, \textit{msgs}))$ 
  within
  ops next_mode : State  $\times$  Set[R_Message]  $\rightarrow$  Mode;
        next_numSTOP : State  $\times$  Set[R_Message]  $\rightarrow$  Nat

```

```

%% Emergency stop mode:
 $\forall s : State; msgs : Set[R\_Message]$ 
•  $Emergency(s, msgs)$ 
 $\Rightarrow next\_mode(s, msgs) = EmergencyStop$ 
%% Normal mode:
•  $\neg Emergency(s, msgs) \wedge Everything\_OK(s, msgs)$ 
 $\Rightarrow next\_mode(s, msgs) = Normal$ 
%% Degraded mode:
•  $\neg Emergency(s, msgs) \wedge \neg Everything\_OK(s, msgs)$ 
 $\wedge PU\_OK(s, msgs, WaterLevel)$ 
 $\wedge Transmission\_OK(s, msgs)$ 
 $\Rightarrow next\_mode(s, msgs) = Degraded$ 
%% Rescue mode:
•  $\neg Emergency(s, msgs) \wedge \neg PU\_OK(s, msgs, WaterLevel)$ 
 $\wedge SystemStillControllable(s, msgs)$ 
 $\wedge Transmission\_OK(s, msgs)$ 
 $\Rightarrow next\_mode(s, msgs) = Rescue$ 
%% next_numSTOP:
•  $next\_numSTOP(s, msgs)$ 
 $= numSTOP(s) + 1$  when STOP eps msgs else 0

spec SBCS_STATE_2 =
  SBCS_STATE_1
then free type
  Status ::= OK | FailureWithoutAck | FailureWithAck
  op   status : State  $\times$  PhysicalUnit  $\rightarrow$  Status

spec STATUS_EVOLUTION
  [pred PU_OK : State  $\times$  Set[R_Message]  $\times$  PhysicalUnit]
  given SBCS_STATE_2 =
  op   next_status
        : State  $\times$  Set[R_Message]  $\times$  PhysicalUnit  $\rightarrow$  Status
 $\forall s : State; msgs : Set[R\_Message]; pu : PhysicalUnit$ 
•  $status(s, pu) = OK \wedge PU\_OK(s, msgs, pu)$ 
 $\Rightarrow next\_status(s, msgs, pu) = OK$ 
•  $status(s, pu) = OK \wedge \neg PU\_OK(s, msgs, pu)$ 
 $\Rightarrow next\_status(s, msgs, pu) = FailureWithoutAck$ 
•  $status(s, pu) = FailureWithoutAck$ 
 $\wedge FAILURE\_ACKNOWLEDGEMENT(pu) \text{ eps } msgs$ 
 $\Rightarrow next\_status(s, msgs, pu) = FailureWithAck$ 
•  $status(s, pu) = FailureWithoutAck$ 
 $\wedge \neg FAILURE\_ACKNOWLEDGEMENT(pu) \text{ eps } msgs$ 
 $\Rightarrow next\_status(s, msgs, pu) = FailureWithoutAck$ 
•  $status(s, pu) = FailureWithAck \wedge REPAIRED(pu) \text{ eps } msgs$ 

```

$\Rightarrow \text{next_status}(s, \text{msgs}, \text{pu}) = \text{OK}$

- $\text{status}(s, \text{pu}) = \text{FailureWithAck}$
- $\wedge \neg \text{REPAIRED}(\text{pu}) \text{ eps } \text{msgs}$
- $\Rightarrow \text{next_status}(s, \text{msgs}, \text{pu}) = \text{FailureWithAck}$

spec MESSAGE_TRANSMISSION_SYSTEM_FAILURE =
SBCS_STATE_2

then local

%% Static analysis:

pred $__ \text{is_static_OK} : \text{Set}[\text{R_Message}]$

$\forall \text{msgs} : \text{Set}[\text{R_Message}]$

- msgs is_static_OK
- $\Leftrightarrow \neg \text{junk eps msgs} \wedge (\exists! v : \text{Value} \bullet \text{LEVEL}(v) \text{ eps msgs})$
- $\wedge (\exists! v : \text{Value} \bullet \text{STEAM}(v) \text{ eps msgs})$
- $\wedge (\forall \text{pn} : \text{PumpNumber}$
 - $\exists! \text{ps} : \text{PumpState}$
 - $\text{PUMP_STATE}(\text{pn}, \text{ps}) \text{ eps msgs}$)
- $\wedge (\forall \text{pn} : \text{PumpNumber}$
 - $\exists! \text{pcs} : \text{PumpControllerState}$
 - $\text{PUMP_CONTROLLER_STATE}(\text{pn}, \text{pcs}) \text{ eps msgs}$)
- $\wedge \forall \text{pu} : \text{PhysicalUnit}$
 - $\neg \text{FAILURE_ACKNOWLEDGEMENT}(\text{pu}) \text{ eps msgs}$
 - $\wedge \text{REPAIRED}(\text{pu}) \text{ eps msgs}$

%% Dynamic analysis:

pred $__ \text{is_NOT_dynamic_OK_for_} __$
 $: \text{Set}[\text{R_Message}] \times \text{State}$

$\forall s : \text{State}; \text{msgs} : \text{Set}[\text{R_Message}]$

- $\text{msgs is_NOT_dynamic_OK_for } s$
- $\Leftrightarrow (\neg \text{mode}(s) = \text{Initialization}$
 - $\wedge (\text{STEAM_BOILER_WAITING eps msgs}$
 - $\vee \text{PHYSICAL_UNITS_READY eps msgs}))$
 - $\vee (\exists \text{pu} : \text{PhysicalUnit}$
 - $\text{FAILURE_ACKNOWLEDGEMENT}(\text{pu}) \text{ eps msgs}$
 - $\wedge (\text{status}(s, \text{pu}) = \text{OK}$
 - $\vee \text{status}(s, \text{pu}) = \text{FailureWithAck}))$
 - $\vee \exists \text{pu} : \text{PhysicalUnit}$
 - $\text{REPAIRED}(\text{pu}) \text{ eps msgs}$
 - $\wedge (\text{status}(s, \text{pu}) = \text{OK}$
 - $\vee \text{status}(s, \text{pu}) = \text{FailureWithoutAck}))$

within

pred $\text{Transmission_OK} : \text{State} \times \text{Set}[\text{R_Message}]$

$\forall s : \text{State}; \text{msgs} : \text{Set}[\text{R_Message}]$

- $\text{Transmission_OK}(s, \text{msgs})$

$$\Leftrightarrow \text{msgs is_static_OK} \\ \wedge \neg \text{msgs is_NOT_dynamic_OK_for } s$$

```

spec SBCS_STATE_3 =
  SBCS_STATE_2
then free type
  ExtendedPumpState ::= sort PumpState | Unknown_PS
  op   PS_predicted
        : State  $\times$  PumpNumber  $\rightarrow$  ExtendedPumpState
        %{
        status(s,Pump(pn)) = OK <=>
        not (PS_predicted(s,pn) = Unknown_PS) }%

spec PUMP_FAILURE =
  SBCS_STATE_3
then pred Pump_OK
        : State  $\times$  Set[R_Message]  $\times$  PumpNumber
 $\forall s : \text{State}; \text{msgs} : \text{Set}[R\_Message]; \text{pn} : \text{PumpNumber}$ 
  • Pump_OK(s, msgs, pn)
     $\Leftrightarrow$  PS_predicted(s, pn) = Unknown_PS
     $\vee$  PUMP_STATE
    (pn, PS_predicted (s, pn) as PumpState) eps msgs

spec SBCS_STATE_4 =
  SBCS_STATE_3
then free type
  ExtendedPumpControllerState
  ::= sort PumpControllerState | SoonFlow | Unknown_PCS
  op   PCS_predicted
        : State  $\times$  PumpNumber  $\rightarrow$  ExtendedPumpControllerState
        %{
        status(s,PumpController(pn)) = OK =>
        not (PCS_predicted(s,pn) = Unknown_PCS) }%

spec PUMP_CONTROLLER_FAILURE =
  SBCS_STATE_4
then pred Pump_Controller_OK
        : State  $\times$  Set[R_Message]  $\times$  PumpNumber
 $\forall s : \text{State}; \text{msgs} : \text{Set}[R\_Message]; \text{pn} : \text{PumpNumber}$ 
  • Pump_Controller_OK(s, msgs, pn)
     $\Leftrightarrow$  PCS_predicted(s, pn) = Unknown_PCS
     $\vee$  PCS_predicted(s, pn) = SoonFlow
     $\vee$  PUMP_CONTROLLER_STATE
    (pn, PCS_predicted (s, pn) as PumpControllerState)

```

eps msgs

```

spec SBCS_STATE_5 =
  SBCS_STATE_4
then free type Valpair ::= pair(low : Value; high : Value)
  ops steam_predicted, level_predicted : State → Valpair
    %{
      low(steam_predicted(s)) is the minimal steam output predicted,
      high(steam_predicted(s)) is the maximal steam output predicted,
      and similarly for level_predicted. }%

```

```

spec STEAM_FAILURE =
  SBCS_STATE_5
then pred Steam_OK : State × Set[R_Message]
  ∀ s : State; msgs : Set[R_Message]
  • Steam_OK(s, msgs)
    ⇔ ∀ v : Value
      • STEAM (v) eps msgs
        ⇒ low (steam_predicted(s)) ≤ v
          ∧ v ≤ high (steam_predicted(s))

```

```

spec LEVEL_FAILURE =
  SBCS_STATE_5
then pred Level_OK : State × Set[R_Message]
  ∀ s : State; msgs : Set[R_Message]
  • Level_OK(s, msgs)
    ⇔ ∀ v : Value
      • LEVEL (v) eps msgs
        ⇒ low (level_predicted(s)) ≤ v
          ∧ v ≤ high (level_predicted(s))

```

```

spec FAILURE_DETECTION =
  {MESSAGE_TRANSMISSION_SYSTEM_FAILURE
  and PUMP_FAILURE
  and PUMP_CONTROLLER_FAILURE
  and STEAM_FAILURE
  and LEVEL_FAILURE
  then pred PU_OK
    : State × Set[R_Message] × PhysicalUnit
  ∀ s : State; msgs : Set[R_Message]; pn : PumpNumber
  • PU_OK(s, msgs, Pump(pn))
    ⇔ Pump_OK(s, msgs, pn)
  • PU_OK(s, msgs, PumpController(pn))
    ⇔ Pump_Controller_OK(s, msgs, pn)

```

```

    •  $PU\_OK(s, msgs, SteamOutput)$ 
       $\Leftrightarrow Steam\_OK(s, msgs)$ 
    •  $PU\_OK(s, msgs, WaterLevel) \Leftrightarrow Level\_OK(s, msgs)$ 
  }
  hide preds Pump_OK, Pump_Controller_OK, Steam_OK,
            Level_OK

```

```

spec STEAM_AND_LEVEL_PREDICTION =
  FAILURE_DETECTION
and SET[sort PumpNumber fit Elem  $\mapsto$  PumpNumber]
then local
  ops received_steam : State  $\times$  Set[R_Message]  $\rightarrow$  Value;
      adjusted_steam : State  $\times$  Set[R_Message]  $\rightarrow$  Valpair;
      received_level : State  $\times$  Set[R_Message]  $\rightarrow$  Value;
      adjusted_level : State  $\times$  Set[R_Message]  $\rightarrow$  Valpair;
      broken_pumps
      : State  $\times$  Set[R_Message]  $\rightarrow$  Set[PumpNumber];
      reliable_pumps
      : State  $\times$  Set[R_Message]  $\times$  PumpState  $\rightarrow$ 
        Set[PumpNumber]
      %% Axioms for STEAM:
       $\forall s : State; msgs : Set[R\_Message]; pn : PumpNumber;$ 
       $ps : PumpState$ 
      •  $Transmission\_OK(s, msgs)$ 
         $\Rightarrow STEAM (received\_steam(s, msgs)) \text{ eps } msgs$ 
      •  $adjusted\_steam(s, msgs)$ 
         $= pair(received\_steam(s, msgs), received\_steam(s, msgs))$ 
        when  $Transmission\_OK(s, msgs)$ 
           $\wedge PU\_OK(s, msgs, SteamOutput)$ 
        else  $steam\_predicted(s)$ 
      %% Axioms for LEVEL:
      •  $Transmission\_OK(s, msgs)$ 
         $\Rightarrow LEVEL (received\_level(s, msgs)) \text{ eps } msgs$ 
      •  $adjusted\_level(s, msgs)$ 
         $= pair(received\_level(s, msgs), received\_level(s, msgs))$ 
        when  $Transmission\_OK(s, msgs)$ 
           $\wedge PU\_OK(s, msgs, WaterLevel)$ 
        else  $level\_predicted(s)$ 
      %% Axioms for auxiliary pumps operations:
      •  $pn \text{ eps } broken\_pumps (s, msgs)$ 
         $\Leftrightarrow \neg PU\_OK(s, msgs, Pump(pn))$ 
           $\wedge PU\_OK(s, msgs, PumpController(pn))$ 
      •  $pn \text{ eps } reliable\_pumps (s, msgs, ps)$ 
         $\Leftrightarrow \neg pn \text{ eps } broken\_pumps (s, msgs)$ 

```

\wedge PUMP_STATE (pn, ps) eps msgs

within

ops *next_steam_predicted*
: State \times Set[R_Message] \rightarrow Valpair;
chosen_pumps
: State \times Set[R_Message] \times PumpState \rightarrow
Set[PumpNumber];
minimal_pumped_water, maximal_pumped_water
: State \times Set[R_Message] \rightarrow Value;
next_level_predicted
: State \times Set[R_Message] \rightarrow Valpair

pred DangerousWaterLevel : State \times Set[R_Message]
%% Axioms for STEAM:

$\forall s : \text{State}; \text{msgs} : \text{Set}[\text{R_Message}]; \text{pn} : \text{PumpNumber}$

- $\text{low}(\text{next_steam_predicted}(s, \text{msgs}))$
 $= \max(0, \text{low}(\text{adjusted_steam}(s, \text{msgs})) - (U2 * dt))$
- $\text{high}(\text{next_steam_predicted}(s, \text{msgs}))$
 $= \min(W, \text{high}(\text{adjusted_steam}(s, \text{msgs})) + (U1 * dt))$

%% Axioms for PUMPS:

- $\text{pn eps chosen_pumps}(s, \text{msgs}, \text{Open})$
 $\Rightarrow \text{pn eps reliable_pumps}(s, \text{msgs}, \text{Closed})$
- $\text{pn eps chosen_pumps}(s, \text{msgs}, \text{Closed})$
 $\Rightarrow \text{pn eps reliable_pumps}(s, \text{msgs}, \text{Open})$
- $\text{minimal_pumped_water}(s, \text{msgs})$
 $= (dt * P) * \#$
 $(\text{reliable_pumps}(s, \text{msgs}, \text{Open}) - \text{chosen_pumps}(s, \text{msgs}, \text{Closed}))$
- $\text{maximal_pumped_water}(s, \text{msgs})$
 $= (dt * P) * \#$
 $((\text{reliable_pumps}(s, \text{msgs}, \text{Open}) \text{ union } \text{chosen_pumps}(s, \text{msgs}, \text{Open}))$
 $\text{ union } \text{broken_pumps}(s, \text{msgs}))$
 $- \text{chosen_pumps}(s, \text{msgs}, \text{Closed}))$

%% Axioms for LEVEL:

- $\text{low}(\text{next_level_predicted}(s, \text{msgs}))$
 $= \max$
 $(0,$
 $(\text{low}(\text{adjusted_level}(s, \text{msgs})) +$
 $\text{minimal_pumped_water}(s, \text{msgs}))$
 $-$
 $((dt \text{ square} * U1 \text{ half}) +$
 $(dt * \text{high}(\text{adjusted_steam}(s, \text{msgs}))))))$
- $\text{high}(\text{next_level_predicted}(s, \text{msgs}))$
 $= \min$

```

(C,
  (high (adjusted_level(s, msgs)) +
    maximal_pumped_water (s, msgs))
  -
  ((dt square * U2 half) +
    (dt * low (adjusted_steam(s, msgs))))))
• DangerousWaterLevel(s, msgs)
  ⇔ low (next_level_predicted(s, msgs)) <= M1
    ∨ M2 <= high (next_level_predicted(s, msgs))
hide ops minimal_pumped_water, maximal_pumped_water

spec PUMP_STATE_PREDICTION =
  STATUS_EVOLUTION[FAILURE_DETECTION]
and STEAM_AND_LEVEL_PREDICTION
then op next_PS_predicted
  : State × Set[R_Message] × PumpNumber →
  ExtendedPumpState
  ∀ s : State; msgs : Set[R_Message]; pn : PumpNumber
  • next_PS_predicted(s, msgs, pn)
  = Unknown_PS
  when ¬ next_status(s, msgs, Pump(pn)) = OK
  else Open
    when (PUMP_STATE (pn, Open) eps msgs
      ∧ ¬ pn eps chosen_pumps (s, msgs, Closed))
      ∨ pn eps chosen_pumps (s, msgs, Open)
    else Closed

spec PUMP_CONTROLLER_STATE_PREDICTION =
  STATUS_EVOLUTION[FAILURE_DETECTION]
and STEAM_AND_LEVEL_PREDICTION
then op next_PCS_predicted
  : State × Set[R_Message] × PumpNumber →
  ExtendedPumpControllerState
  ∀ s : State; msgs : Set[R_Message]; pn : PumpNumber
  • next_PCS_predicted(s, msgs, pn)
  = Unknown_PCS
  when ¬ next_status(s, msgs, PumpController(pn)) = OK
    ∧ next_status(s, msgs, Pump(pn)) = OK
  else Flow
    when (PUMP_CONTROLLER_STATE (pn, Flow)
      eps msgs
      ∨ (PUMP_CONTROLLER_STATE
        (pn, NoFlow) eps msgs
        ∧ PCS_predicted(s, pn) = SoonFlow))

```

```

       $\wedge \neg pn \text{ eps chosen\_pumps } (s, \text{msgs}, \text{Closed})$ 
    else NoFlow
      when  $pn \text{ eps chosen\_pumps } (s, \text{msgs}, \text{Closed})$ 
         $\vee (\text{PUMP\_CONTROLLER\_STATE}$ 
           $(pn, \text{NoFlow}) \text{ eps msgs}$ 
           $\wedge \neg \text{PCS\_predicted}(s, pn) = \text{SoonFlow}$ 
           $\wedge \neg pn \text{ eps chosen\_pumps}$ 
             $(s, \text{msgs}, \text{Open}))$ 
        else SoonFlow

```

end

```

spec PU_PREDICTION =
  PUMP_STATE_PREDICTION
and PUMP_CONTROLLER_STATE_PREDICTION
end

```

```

spec SBCS_ANALYSIS =
  MODE_EVOLUTION[PU_PREDICTION]
then local
  ops PumpMessages, FailureDetectionMessages
    : State  $\times$  Set[R_Message]  $\rightarrow$  Set[S_Message];
    RepairedAcknowledgementMessages
    : Set[R_Message]  $\rightarrow$  Set[S_Message]
   $\forall s : \text{State}; \text{msgs} : \text{Set}[R\_Message]; \text{Smsg} : S\_Message$ 
  • Smsg eps PumpMessages (s, msgs)
     $\Leftrightarrow \exists pn : \text{PumpNumber}$ 
    •  $(pn \text{ eps chosen\_pumps } (s, \text{msgs}, \text{Open})$ 
       $\wedge \text{Smsg} = \text{OPEN\_PUMP}(pn)$ 
       $\vee (pn \text{ eps chosen\_pumps } (s, \text{msgs}, \text{Closed})$ 
         $\wedge \text{Smsg} = \text{CLOSE\_PUMP}(pn))$ 
    • Smsg eps FailureDetectionMessages (s, msgs)
       $\Leftrightarrow \exists pu : \text{PhysicalUnit}$ 
      •  $\text{Smsg} = \text{FAILURE\_DETECTION}(pu)$ 
         $\wedge \text{next\_status}(s, \text{msgs}, pu) = \text{FailureWithoutAck}$ 
    • Smsg eps RepairedAcknowledgementMessages (msgs)
       $\Leftrightarrow \exists pu : \text{PhysicalUnit}$ 
      •  $\text{Smsg} = \text{REPAIRED\_ACKNOWLEDGEMENT}(pu)$ 
         $\wedge \text{next\_status}(s, \text{msgs}, pu) = \text{FailureWithAck}$ 
  within
  op messages_to_send
    : State  $\times$  Set[R_Message]  $\rightarrow$  Set[S_Message]
   $\forall s : \text{State}; \text{msgs} : \text{Set}[R\_Message]$ 
  • messages_to_send(s, msgs)
    = ((PumpMessages (s, msgs) union

```

```

    FailureDetectionMessages (s, msgs))
    union RepairedAcknowledgementMessages (msgs))
    + MODE (next_mode(s, msgs))
end

spec SBCS_STATE =
  PRELIMINARY
then sort State
  free type
    Status ::= OK | FailureWithoutAck | FailureWithAck
  free type
    ExtendedPumpState ::= sort PumpState | Unknown_PS
  free type
    ExtendedPumpControllerState
    ::= sort PumpControllerState | SoonFlow | Unknown_PCS
  free type Valpair ::= pair(low : Value; high : Value)
  ops mode : State → Mode;
    numSTOP : State → Nat;
    status : State × PhysicalUnit → Status;
    PS_predicted
    : State × PumpNumber → ExtendedPumpState;
    PCS_predicted
    : State × PumpNumber →
      ExtendedPumpControllerState;
    steam_predicted, level_predicted : State → Valpair

spec STEAM_BOILER_CONTROL_SYSTEM =
  SBCS_ANALYSIS
then op init : State
  pred is_step
    : State × Set[R_Message] × Set[S_Message] × State
    %% Specification of the initial state init:
    • mode(init) = Normal ∨ mode(init) = Degraded
    %% Specification of is_step:
    ∀ s, s' : State; msgs : Set[R_Message];
    Smsg : Set[S_Message]
    • is_step(s, msgs, Smsg, s')
      ⇔ mode(s') = next_mode(s, msgs)
        ∧ numSTOP(s') = next_numSTOP(s, msgs)
        ∧ (∀ pu : PhysicalUnit
          • status(s', pu) = next_status(s, msgs, pu))
        ∧ (∀ pn : PumpNumber
          • PS_predicted(s', pn)
            = next_PS_predicted(s, msgs, pn)

```

```

      ^ PCS_predicted(s', pn)
      = next_PCS_predicted(s, msgs, pn)
    ^ steam_predicted(s') = next_steam_predicted(s, msgs)
    ^ level_predicted(s') = next_level_predicted(s, msgs)
    ^ Smsg = messages_to_send(s, msgs)
  then
    %% Specification of the reachable states:
    free
    {pred reach : State
    ∀ s, s' : State; msgs : Set[R_Message];
    Smsg : Set[S_Message]
    • reach(init)
    • reach(s) ^ is_step(s, msgs, Smsg, s') ⇒ reach(s')}

  arch spec ARCH_SBCS =
    units P : VALUE → PRELIMINARY;
    S : PRELIMINARY → SBCS_STATE;
    A : SBCS_STATE → SBCS_ANALYSIS;
    C : SBCS_ANALYSIS → STEAM_BOILER_CONTROL_SYSTEM
    result lambda V : VALUE • C [A [S [P [V]]]]

  arch spec ARCH_PRELIMINARY =
    units SET : {sort Elem} × NAT → SET[sort Elem];
    B : BASICS;
    MS : MESSAGES_SENT given B;
    MR : VALUE → MESSAGES_RECEIVED given B;
    CST : VALUE → SBCS_CONSTANTS
    result λ V : VALUE
    • SET [MS fit Elem ↦ S_Message] [V]
      and SET [MR [V] fit Elem ↦ R_Message] [V]
      and CST [V]

  spec SBCS_STATE_IMPL =
    PRELIMINARY
  then free type
    Status ::= OK | FailureWithoutAck | FailureWithAck
    free type
    ExtendedPumpState ::= sort PumpState | Unknown_PS
    free type
    ExtendedPumpControllerState
    ::= sort PumpControllerState | SoonFlow | Unknown_PCS
    free type Valpair ::= pair(low : Value; high : Value)
  then TOTALMAP[BASICS fit S ↦ PhysicalUnit][sort Status]

```

```

and TOTALMAP
  [BASICS fit  $S \mapsto \text{PumpNumber}$ ] [sort ExtendedPumpState]
and TOTALMAP
  [BASICS fit  $S \mapsto \text{PumpNumber}$ ]
  [sort ExtendedPumpControllerState]
then free type
  State
  ::= mk_state(mode : Mode;
               numSTOP : Nat;
               status : TotalMap[PhysicalUnit,Status];
               PS_predicted :
               TotalMap[PumpNumber,ExtendedPumpState];
               PCS_predicted :
               TotalMap
               [PumpNumber,ExtendedPumpControllerState];
               steam_predicted, level_predicted : Valpair)
  ops status(s : State; pu : PhysicalUnit) : Status
      = lookup(pu, status(s));
  PS_predicted
  (s : State; pn : PumpNumber) : ExtendedPumpState
  = lookup(pn, PS_predicted(s));
  PCS_predicted
  (s : State; pn : PumpNumber)
  : ExtendedPumpControllerState
  = lookup(pn, PCS_predicted(s))

unit spec UNIT_SBCS_STATE =
  PRELIMINARY  $\rightarrow$  SBCS_STATE_IMPL

arch spec ARCH_ANALYSIS =
  units FD : SBCS_STATE  $\rightarrow$  FAILURE_DETECTION;
  PR : FAILURE_DETECTION  $\rightarrow$  PU_PREDICTION;
  ME : PU_PREDICTION  $\rightarrow$  MODE_EVOLUTION[PU_PREDICTION];
  MTS : MODE_EVOLUTION[PU_PREDICTION]  $\rightarrow$  SBCS_ANALYSIS
  result lambda  $S$  : SBCS_STATE • MTS [ME [PR [FD [S]]]]

arch spec ARCH_FAILURE_DETECTION =
  units
  MTSF : SBCS_STATE  $\rightarrow$  MESSAGE_TRANSMISSION_SYSTEM_FAILURE;
  PF : SBCS_STATE  $\rightarrow$  PUMP_FAILURE;
  PCF : SBCS_STATE  $\rightarrow$  PUMP_CONTROLLER_FAILURE;
  SF : SBCS_STATE  $\rightarrow$  STEAM_FAILURE;
  LF : SBCS_STATE  $\rightarrow$  LEVEL_FAILURE;
  PU :

```

MESSAGE_TRANSMISSION_SYSTEM_FAILURE ×
 PUMP_FAILURE × PUMP_CONTROLLER_FAILURE ×
 STEAM_FAILURE × LEVEL_FAILURE → FAILURE_DETECTION

result

lambda *S* : SBCS_STATE

• PU [MTSF [S]] [PF [S]] [PCF [S]] [SF [S]] [LF [S]]

hide *Pump_OK*, *Pump_Controller_OK*, *Steam_OK*,
Level_OK

arch spec ARCH_PREDICTION =

units

SE :

FAILURE_DETECTION → STATUS_EVOLUTION[FAILURE_DETECTION];

SLP : FAILURE_DETECTION → STEAM_AND_LEVEL_PREDICTION;

PP :

STATUS_EVOLUTION[FAILURE_DETECTION] ×

STEAM_AND_LEVEL_PREDICTION →

PUMP_STATE_PREDICTION;

PCP :

STATUS_EVOLUTION[FAILURE_DETECTION] ×

STEAM_AND_LEVEL_PREDICTION →

PUMP_CONTROLLER_STATE_PREDICTION

result

lambda *FD* : FAILURE_DETECTION

• **local** SEFD = SE [FD]; SLPFD = SLP [FD] **within**

{PP [SEFD] [SLPFD] **and** PCP [SEFD] [SLPFD]}

unit spec SBCS_OPEN = VALUE → STEAM_BOILER_CONTROL_SYSTEM

refinement REF_SBCS = SBCS_OPEN **refined to arch spec** ARCH_SBCS

unit spec STATEABSTR = PRELIMINARY → SBCS_STATE

refinement STATeref = STATEABSTR **refined to** UNIT_SBCS_STATE

refinement REF_SBCS' = REF_SBCS **then**

{P **to arch spec** ARCH_PRELIMINARY, S **to** STATeref,
 A **to arch spec** ARCH_ANALYSIS}

refinement REF_SBCS'' = REF_SBCS' **then**

{A **to**

{FD **to arch spec** ARCH_FAILURE_DETECTION,

PR **to arch spec** ARCH_PREDICTION }}

