A Framework for Model-based Testing of Integrated Modular Avionics

Dissertation zur Erlangung des
Doktorgrades der Ingenieurwissenschaften
(Dr.-Ing.)

vorgelegt von
Christof Efkemann

Fachbereich 3
Informatik/Mathematik

Universität Bremen

A Framework for Model-based Testing of Integrated Modular Avionics

Dissertation zur Erlangung des
Doktorgrades der Ingenieurwissenschaften
(Dr.-Ing.)

vorgelegt von
Christof Efkemann

im Fachbereich 3 (Informatik/Mathematik)
der Universität Bremen
im August 2014

Revision 205.
Typeset with XƎLaTeX in Adobe Garamond Pro.

Eingereicht am 06. August 2014

Gutachter:
Prof. Dr. Jan Peleska (Universität Bremen)
Prof. Dr. Anne E. Haxthausen (Technical University of Denmark)

Erklärung

Die vorliegende Dissertation wurde ohne fremde Hilfe angefertigt. Es wurden keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Alle Stellen, die wörtlich aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht.

Datum	Unterschrift

Zusammenfassung

In modernen Flugzeugen basiert das Design der Elektronik und der Steuerungssysteme auf dem Konzept der Integrierten Modularen Avionik (IMA). Obwohl dies diverse Vorteile mit sich bringt (Gewichtsreduktion, niedrigerer Strom- und Treibstoffverbrauch, geringere Entwicklungs- und Zertifizierungskosten und -aufwände), stellt die IMA-Plattform eine zusätzliche Komplexitätsebene dar.

Aufgrund der sicherheitskritischen Eigenschaften vieler Avionikfunktionen sind sorgfältige und akkurate Verifikation und Tests dieser Systeme zwingend erforderlich.

Diese Dissertation beschreibt Forschungsergebnisse bezüglich des modellbasierten Testens von IMA-Systemen, welche zum Teil im Rahmen des europäischen Forschungsprojekts SCARLETT erzielt wurden. Die Arbeit beschreibt ein vollständiges Rahmenwerk, welches es einem IMA-Domänenexperten ermöglicht, modellbasierte Tests auf Modul-, Konfigurations- sowie Anwendungsebene zu entwerfen und in einer standardisierten Testumgebung auszuführen.

Der erste Teil dieser Arbeit bietet Hintergrundinformationen zu den ihr zugrunde liegenden Themen: dem IMA-Konzept, domänenspezifischen Sprachen, modellbasiertem Testen und dem TTCN-3-Standard. Im zweiten Teil werden das Rahmenwerk der IMA-Testmodellierungssprache (ITML) und dazugehörige Komponenten eingeführt. Es wird eine hierfür maßgeschneiderte TTCN-3-Testumgebung mit passenden Adaptern und Codecs beschrieben. Basierend auf MetaEdit+ und dem Meta-Metamodell GOPPRR werden die drei Varianten der domänenspezifischen Sprache ITML definiert, jeweils mit abstrakter und konkreter Syntax sowie statischer und dynamischer Semantik. Die Generierung von Testprozeduren aus ITML-Modellen wird im Detail erklärt. Darüber hinaus werden das Design und die Entwicklung eines universellen Testagenten beschrieben. Zur Steuerung des Agenten wird ein dediziertes Kommunikationsprotokoll definiert.

Für den dritten Teil dieser Arbeit wurde eine Evaluation des Rahmenwerks durchgeführt. Es werden Einsatzszenarien im SCARLETT-Projekt beschrieben und Vergleiche zwischen ITML und verwandten Werkzeugen und Ansätzen gezogen. Außerdem wird auf die Vorteile der Benutzung des ITML-Rahmenwerks durch IMA-Domänenexperten eingegangen.

Der letzte Teil bietet eine Reihe von ITML-Beispielmodellen. Er enthält darüber hinaus Referenzmaterial, u. a. XML-Schemata, Quellcode des Rahmenwerks und Modellvalidatoren.

Abstract

In modern aircraft, electronics and control systems are designed based on the Integrated Modular Avionics (IMA) system architecture. While this has numerous advantages (reduction of weight, reduced power and fuel consumption, reduction of development cost and certification effort), the IMA platform also adds an additional layer of complexity.

Due to the safety-critical nature of many avionics functions careful and accurate verification and testing are imperative.

This thesis describes results achieved from research on model-based testing of IMA systems, in part obtained during the European research project SCARLETT. It presents a complete framework which enables IMA domain experts to design and run model-based tests on bare module, configured module, and application level in a standardised test environment.

The first part of this thesis provides background information on the relevant topics: the IMA concept, domain-specific languages, model-based testing, and the TTCN-3 standard. The second part introduces the IMA Test Modelling Language (ITML) framework and its components. It describes a tailored TTCN-3 test environment with appropriate adapters and codecs. Based on MetaEdit+ and its meta-metamodel GOPPRR, it defines the three variants of the domain-specific language ITML, each with its abstract and concrete syntax as well as static and dynamic semantics. The process of test procedure generation from ITML models is explained in detail. Furthermore, the design and implementation of a universal Test Agent is shown. A dedicated communication protocol for controlling the agent is defined as well.

The third part provides an evaluation of the framework. It shows usage scenarios in the SCARLETT project, gives a comparison to related tools and approaches, and explains the advantages of using the ITML framework for an IMA domain expert.

The final part presents several example ITML models. It also provides reference material like XML schemata, framework source code, and model validators.

Acknowledgements

Writing this thesis would not have been possible if it weren't for the support of a number of people. First of all, I would like to thank my advisor, Jan Peleska, for giving me the opportunity to work in his research group at the University of Bremen. Second, I am very grateful to have had the opportunity of being a part of the European research project SCARLETT. I enjoyed these three and a half years of my life, working with such a vast group of interesting and skilled people from all across Europe. Two people I would like to thank in particular: Florian Bommer for his strenuous efforts to make the I/O-intensive demonstration a success, and Arne Graepel for his very helpful support with the TTCN-3 test environment.

I would also like to thank my kind colleagues, both at the university as well as at Verified Systems. I very much enjoyed the pleasant working atmosphere and their constructive support. In particular, I would like to thank Uwe Schulze and Helge Löding for sharing their invaluable expertise on model-based testing and rtt-tcgen.

Not unmentioned shall go my two proofreaders: Oliver Meyer and Kai Thomsen. Thanks to their labour many a typo or inaccuracy could be eliminated—any remaining are entirely my own fault.

And last, but by no means least, I would like to thank my family and my friends for their love, their trust, and for believing in me.

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

In today's world we find ourselves more and more surrounded by electronic devices. Laptop computers, smartphones, or satellite navigation systems are a part of our everyday life. They are useful tools and make our lives easier, provided they work properly. A malfunction of such a device is usually a mere nuisance.

On the other hand, electronic devices are employed in *safety-critical systems* like aircraft engine controllers, anti-lock braking systems, electronic signal boxes, and medical life-support systems. Here, a malfunction or failure has potentially catastrophic consequences and can result in loss of lives. Assuring the safety and reliability of such systems is therefore crucial.

In particular, the *Integrated Modular Avionics* (IMA) architecture provides an example of such a system from the aerospace domain. IMA modules can host crucial aircraft functions like flight control, fire and smoke detection, or braking control systems. A *partitioning* scheme allows for hosting of several aircraft functions safely on the same module. Apart from having numerous advantages, this architecture also comes with a higher level of integration and increased complexity. This makes the development of such systems an interesting and also very important research subject.

Domain-specific modelling (DSM) is a relatively new methodology that allows elements and concepts from a specific application domain to be used. In contrast to general-purpose modelling (e. g. UML), DSM is more intuitive in its application and allows a higher level of abstraction. This brings great advantages in usability as well as efficiency.

In *Model-based testing* (MBT), models are used as a means to specify or derive test cases. The goal is to generate executable test procedures automatically instead of writing them by hand. Models can represent nominal System Under Test (SUT) behaviour or desired testing strategies.

The *Testing and Test Control Notation Version 3* (TTCN-3) is a set of standards defining a language for test specification and the infrastructure of a runtime environment for the execution of tests. It is widely used in the area of communication testing and protocol testing. A major benefit of TTCN-3 is the *abstraction* it provides from concrete test environment hardware and interfaces.

Due to their complexity, testing of IMA modules requires considering different levels of module testing, from low-level *bare module tests* to *configured module tests* and *functional tests* (cf. section 2.4 for details).

1.1 Goals

The main goal of this work is to provide a framework for model-based testing of Integrated Modular Avionics systems. The purpose of this framework is to provide the prospective IMA test engineer with a state-of-the-art toolbox that will support him during the design and execution of test suites for IMA-based systems.

1.2 Main Contribution

In order to accomplish this goal, the following main contributions are presented in this dissertation:

1. A domain-specific modelling language specifically designed for the testing of IMA systems, called *IMA Test Modelling Language* (ITML)
2. A flexible and universal *Test Agent* (TA)
3. A TTCN-3-based test execution environment

The combination of these features constitutes a novel approach to IMA testing. They enable test designers to work on a higher level of abstraction, using elements of the application domain instead of concepts of a particular testing tool. They also allow them to specify test cases for bare and configured module tests in a very compact manner, without explicitly referring to the availability and distribution of system resources in the IMA module under test. A generator produces the concrete test procedure in TTCN-3 syntax. This procedure stimulates and controls distributed interactions between test agents on different partitions and modules. For this purpose, the generator evaluates both the test case specification and the IMA module configuration (which can also generated automatically under certain circumstances) and selects the concrete resources involved in compliance with the configuration.

For hardware/software integration tests with the application layer present in each IMA module which is part of the system under test (SUT), behavioural test models based on timed state machines can be used, and the generator derives test cases, test data and test procedures from these models by means of a constraint solver.

A summary of parts of the research presented in this thesis has been published in [EP11].

1.3 Related Work

Experiences in the field of bare and configured module testing result from earlier work in the VICTORIA research project [OH05, Ott07]. In this project, bare module tests were developed based on a library of formal Timed CSP specification templates [Sch00, Pel02]. Unfortunately, this approach requires both a substantial amount of manual programming effort as well as a deep understanding of CSP. As a result, domain experts without knowledge in the field of process algebras were unable to analyse the resulting test procedures. In contrast to that, the methodology presented here uses an intuitive graphic template formalism to describe patterns from which concrete test cases and test data can be derived and which are automatically converted into executable test procedures.

Moreover, the approach chosen in the VICTORIA project relied on specific test engine hardware and test execution software. Conversely, this work shows a method to achieve independence from specific test hardware by deploying a TTCN-3-based test environment [ETS]. The underlying techniques for model-based functional testing are described in more detail in [PVL11], where also references to competing approaches are given.

Domain-specific approaches to test automation are also promising in several other areas: In [Mew10], for example, Kirsten Mewes describes patterns for automated test generation in the railway control system domain, while in [Feu13], Johannes Feuser uses a domain-specific language to create an open model of ETCS. Timed Moore Automata (cf. [LP10]) can be used to model control logic of level crossing systems, on which model-checking and automatic test case generation can be performed. In [EH08] the authors describe a systematic approach for diagnosing errors in the context of hardware/software integration testing and explain this technique with an example from the avionics domain.

[PHL+11] gives an example of practical model-based testing efforts in the automotive domain, where in a collaborative effort from Daimler, Verified Systems, and University of Bremen, a benchmarking method on real-world models is proposed.

1.4 Outline of this Document

This document is divided into four parts. The first part will familiarise the reader with the necessary background information concerning the topics covered in this dissertation. This includes an in-depth discussion of the application domain as required for the next chapters as well as an introduction to the relevant methodologies and concepts as employed in the framework. General information on avionics systems and their development will not be discussed here, as this has already been done extensively, e. g. in [Ott07, ch. 1].

The second part provides an in-depth examination of the individual components that make up the model-based IMA testing framework. It describes the universal *IMA Test Agent* and its control protocol. The structure of the TTCN-3-based *test execution environment* is shown. Afterwards, the domain-specific modelling language for the testing of IMA systems, called *IMA Test Modelling Language*, is presented.

The third part provides an evaluation of the framework. It discusses usage examples, provides a comparison with other tools and approaches, and gives a comprehensive conclusion of the results elaborated in this thesis.

The fourth part provides a collection of detailed materials, including models and source code, for the individual components introduced in the second part.

PART I

Background

CHAPTER 2

Integrated Modular Avionics

The traditional *federated* aircraft controller architecture [Fil03, p. 4] consists of a large number of different, specialised electronics devices. Each of them is dedicated to a special, singular purpose (e. g. flight control, or fire and smoke detection) and has its own custom sensor/actuator wiring. Some of them are linked to each other with dedicated data connections. In the *Integrated Modular Avionics* (IMA) architecture this multitude of device types is replaced by a small number of modular, general-purpose component variants whose instances are linked by a high-speed data network. Due to high processing power each module can host several avionics functions, each of which previously required its own controller. The IMA approach has several main advantages:

- Reduction of weight through a smaller number of physical components and reduced wiring, thereby increasing fuel efficiency.
- Reduction of on-board power consumption by more effective use of computing power and electrical components.
- Lower maintenance costs by reducing the number of different types of replacement units needed to keep on stock.
- Reduction of development costs by provision of a standardised operating system, together with standardised drivers for the avionics interfaces most widely used.
- Reduction of certification effort and costs via incremental certification of hard- and software.

An important aspect of module design is *segregation*: In order to host applications of different safety assurance levels on the same module, it must be ensured that those applications cannot interfere with each other. Therefore a module must support resource partitioning via memory access protection, strict deterministic scheduling and I/O access permissions. Bandwidth limitations on the data network have to be enforced as well.

The standard aircraft documentation reference for IMA is ATA chapter 42. The IMA architecture is currently in use in the Airbus A380, A400M, the future A350XWB, and Boeing 787 Dreamliner aircraft. Predecessors of this architecture can be found in so-called fourth-generation jet fighter aircraft like the Dassault Rafale.

2.1 AFDX

A data network is required for communication between (redundant) IMA modules as well as other hardware. This role is fulfilled by the *Avionics Full DupleX Switched Ethernet* (AFDX) network. It is an implementation of the ARINC specification 664 [Aer09] and is used as high-speed communication link between aircraft controllers. It is the successor of the slower ARINC 429 networks [Aer04].

AFDX is based on 100 Mbit/s Ethernet over twisted-pair copper wires (IEEE 802.3u, 100BASE-TX). This means it is compatible with COTS Ethernet equipment on layers 1 and 2 (physical layer and link layer). Ethernet by itself is not suitable for real-time applications as its timing is not deterministic. Therefore AFDX imposes some constraints in order to achieve full determinism and hard real-time capability.

In AFDX, so called *Virtual Links* (VLs) are employed for bandwidth allocation and packet routing. Each VL has a 16-bit ID, which is encoded into the destination MAC address of each frame sent through this VL. For each VL, only one end system can send frames, while there can be one or more receivers (unidirectional multicast communication, similar to ARINC 429). AFDX switches use a pre-determined configuration to deliver frames based on their VL ID to a set of receiving end systems.

Each VL is allocated a part of the full bandwidth of an AFDX link. To that end, each VL has two attributes: a maximum frame length (L_{max}) in bytes and a bandwidth allocation gap (BAG). The BAG value represents the minimum interval (in milliseconds) between two frames on that VL. Thus, the maximum usable bandwidth in bit/s of a VL can be calculated as:

$$b_{max} = L_{max} \cdot 8 \cdot 1000/BAG$$

End systems use a VL scheduler to ensure minimum latency and jitter for each VL.

Figure 2.1 shows the composition of a complete AFDX frame. The UDP protocol is used on the transport layer, since protocols which rely on retransmission (e. g. TCP) are inadequate in a deterministic network. The message payload is organised into one or more Functional Data Sets (FDS). Each

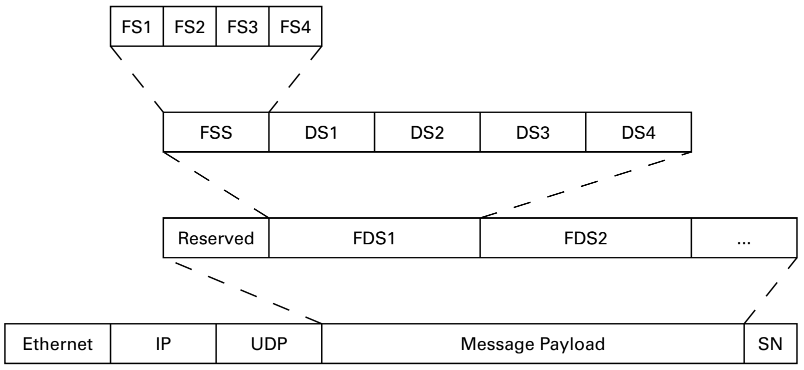


Figure 2.1: AFDX frame

FDS comprises a functional status set (FSS) and up to four data sets (DS). The FSS is split into four functional status (FS) fields. Each FS field determines the validity of the data contained in a data set, similar to the SSM field of ARINC 429 labels. Data sets contain one or more primitive values, like integers, floats, or booleans.

In order to increase reliability, an aircraft data network consists of two independent switched networks. AFDX frames are sent on both networks. If no frame is lost, the other end systems will receive two frames. In order to identify matching frames sent over the redundant links, the message payload is followed by a sequence number field. Of two frames received on different networks with an identical sequence number, only the first is passed up the protocol stack.

2.2 ARINC 653

As indicated before, an IMA module can host multiple avionics functions. The interface between avionics applications and the module's operating system conforms to a standardised API which is defined in the ARINC specification 653 [Aer05].

The system architecture of an IMA module is depicted in figure 2.2. A real-time operating system kernel constitutes the central component. It uses a driver layer for access to the module's I/O hardware (either solely AFDX, or other interfaces like Discretes, CAN, and ARINC 429 buses as well, depending on module type) [Aer05, p. 11].

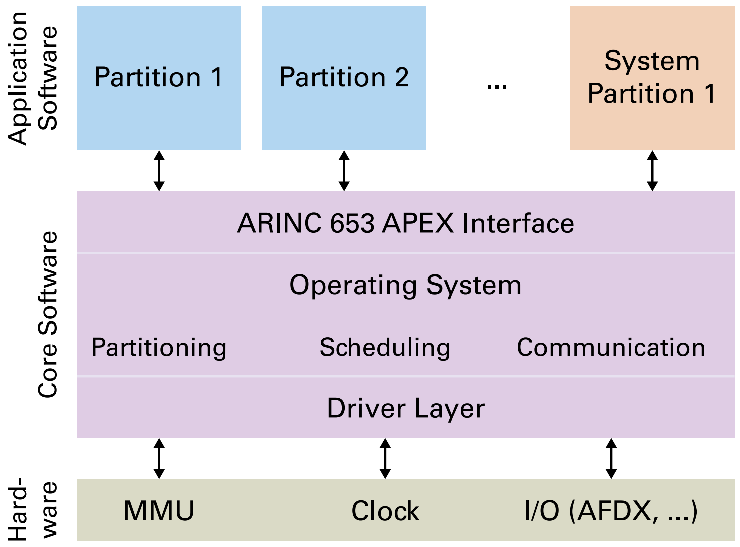


Figure 2.2: IMA module system architecture

2.2.1 Partitioning

In order to guarantee the same isolation of avionics functions residing on a shared module as a federated architecture would provide, it is the operating system's responsibility to implement a concept called *partitioning*. According to [RTC01], a partitioning implementation should comply with the following requirements:

- A software partition should not be allowed to contaminate another partition's code, I/O, or data storage areas.
- A software partition should be allowed to consume shared processor resources only during its period of execution.
- A software partition should be allowed to consume shared I/O resources only during its period of execution.
- Failures of hardware unique to a software partition should not cause adverse effects on other software partitions.
- Software providing partitioning should have the same or higher software level[1] than the highest level of the partitioned software applications.

[1]as defined in [RTC92]

Table 2.1: ARINC 653 communication methods

	Intra-partition	**Inter-partition**
Queuing	Buffer	Queuing Port
Non-Queuing	Blackboard	Sampling Port
Synchronisation	Semaphore Event	—

On the IMA platform, a partition is a fixed set of the module's resources to be used by an avionics application. In particular, each partition is assigned a portion of the module's memory. The operating system ensures that other partitions can neither modify nor access the memory of a partition, similar to memory protection in a UNIX-like operating system. Each partition also receives a fixed amount of CPU time. The operating system's scheduler ensures that no partition can spend CPU time allotted to another partition [Aer05, p. 13].

Two kinds of partitions reside on a module:

Application partitions contain application code that makes up (part of) the implementation of an avionics function.

System partitions on the other hand provide additional module-related services like data loading or health monitoring.

Inside a partition there can be multiple threads of execution, called *processes*. Similar to POSIX threads, all processes within a partition share the resources allocated to the partition. Each process has a priority. A process with a higher priority pre-empts any processes with a lower priority. ARINC 653 defines a set of states a process can be in (Dormant, Waiting, Ready, Running) as well as API functions for process creation and management [Aer05, pp. 18–25].

2.2.2 Communication

The operating system must provide a set of different methods of communication. All of them fall into either of two categories: intra-partition or inter-partition communication. Intra-partition communication always happens between processes of the same partition, while inter-partition communication happens between processes of different partitions or even partitions on different modules. Table 2.1 provides an overview of different communication methods.

A *buffer* is a communication object used to send or receive messages between processes in the same partition. The messages are queued in FIFO order. Messages can have variable size, but must not exceed the maximum message size specified at buffer creation. The maximum number of messages a buffer can hold is defined at buffer creation. When a process reads from a buffer, the oldest message is removed from it and returned to the process [Aer05, p. 36]. A *blackboard* is a similar communication object, but it does not queue messages. Instead, each new message overwrites the current one. A message can be read an arbitrary number of times by any process of the partition, until it is either cleared or overwritten [Aer05, p. 37].

Semaphores and *events* are communication objects that cannot be used to transport messages, but instead serve as synchronisation objects for processes. A semaphore has a counter. The operating system provides functions to increment and decrement the counter. If a process tries to decrement the counter while it is already at zero, the process is forced to wait until another process increments the counter [Aer05, p. 37]. An event object has a list of processes waiting for the event. As soon as the event is set to its signalled state, all waiting processes become ready again [Aer05, p. 38].

Sampling ports and *queuing ports* are used for inter-partition communication. A port is unidirectional (either Source or Destination). Messages sent over a port can have variable size, but must not exceed the defined maximum message size of the port. Ports as well as the communication channels they are connected to are defined at module configuration time. The application cannot influence the routing of messages at runtime. Similar to a buffer, a queuing port stores incoming messages up to a defined maximum number. Each message is removed from the port as it is read. When writing multiple messages to a queuing port, all of the messages will be sent in FIFO order. A sampling port, on the other hand, does not queue multiple messages, neither incoming nor outgoing. Instead, only the latest message is kept [Aer05, pp. 31–35].

2.2.3 Initialisation

All the aforementioned communication objects as well as processes cannot be created during normal operation of a partition. Instead, they must be created during a special initialisation phase. During this phase an initialisation process is executed. Its task is the creation and initialisation of all required resources (processes, ports, buffers, semaphores, etc.) for the partition. Only when this phase is complete and the partition has switched to normal operation mode will the other processes start their execution. For that purpose ARINC 653 describes a set of states a partition can be in, as well as API functions for partition management [Aer05, pp. 15–17].

2.3 Module Configuration

As indicated previously, an integral part of an IMA module is its configuration. In order to be suitable as a general-purpose platform for avionics functions it must be highly configurable to its specific scope of application. This makes the process of creating a configuration a matter of high complexity. In particular, multiple roles are involved in the process: Global module configuration data is managed by the module integrator[2], while partition-specific configuration data are provided by the function suppliers[3] responsible for the implementation of each avionics function.

A complete module configuration consists of a set of data table files—a so-called Interface Control Document (ICD). The module integrator processes the ICD with a configuration tool to generate a binary representation, which is then loaded into the module along with the application software.

The global part of the configuration consists of tables covering the following module-global aspects:

- Physical RAM/ROM sizes and addresses
- CPU cache settings
- Scheduling: Minor Frame duration and number
- Health Monitoring recovery actions
- AFDX VL definitions
- Properties of other I/O hardware (if available)

Each partition has its own set of partition-specific configuration tables. They cover the following aspects of partition configuration:

- Memory allocation (code, data)
- Scheduling: Time slices assignment
- Definition of AFDX messages
- Definition of sampling and queuing ports
- Assignment of ports to communication channels (module-internal or via AFDX)
- Definition of messages for other message-based I/O hardware (if available, e. g. CAN or ARINC 429 bus)

The properties defined in partition-specific configuration are valid for one partition only and cannot have influence on other partitions. As there are cross-references between global and partition-specific configuration tables, it is the module integrator's responsibility to keep the overall configuration consistent and valid (at best using appropriate tool support).

[2]The airframer

[3]The airframer's subcontractors

2.4 Testing IMA Systems

As described above, IMA components have a complex configuration. The fact that it covers many aspects like partitioning, resource allocation, scheduling, I/O configuration, network configuration and internal health monitoring makes the configuration an integral part of the system, and it must be taken into consideration during verification and certification [RTC92, RTC05]. As a result of these considerations, different levels of testing for IMA modules have been defined in a previous research project [Ott07]:

Bare Module Tests are designed to test the module and its operating system at API level. The test cases check for correct behaviour of API calls, while robustness test cases try to violate segregation rules and check that these violation efforts do not succeed. Bare module tests are executed with specialised module configurations designed for these application-independent test objectives. The application layer is substituted by test agents that perform the stimulations on the operating system as required by the testing environment and relay API output data to the testing environment for further inspection.

Configured Module Tests do not focus on the module and its operating system, but are designed to test application-specific configurations meant to be used in the actual aircraft. The test cases check that configured I/O interfaces are usable as defined. Again, the application layer is replaced by test agents.

Functional Tests run with the actual application layer integrated in the module and check for the behavioural correctness of applications as integrated in the IMA module.

2.5 Research Project SCARLETT

SCARLETT (SCAlable & ReconfigurabLe elEctronics plaTforms and Tools) is a European research and technology project [SCA]. With 38 partner organisations (airframers, large industrial companies, SMEs and universities) it is a large-scale integrating project funded under the Seventh Framework Programme [FP7] of the European Commission, submitted to the Transport Call under the Area of Activity 7.1.3 – Ensuring Customer Satisfaction and Safety (Level 2). SCARLETT is a successor to several earlier European research projects in the field of avionics, like PAMELA, NEVADA and VICTORIA. Its goal is the development of a new generation of IMA (IMA2G) with increased scalability, adaptability and fault-tolerance, called *Distributed Modular Electronics (DME)*.

The DME concept aims at the separation of processing power from sensor/ actuator interfaces, thereby reducing the number of different component

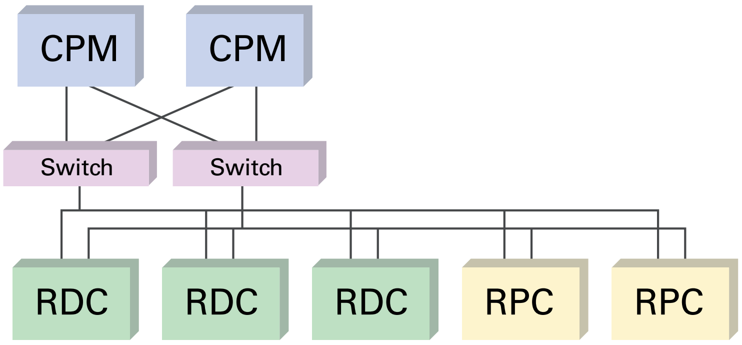


Figure 2.3: DME Architecture

types to a minimum. This also makes DME suitable for a wider range of aircraft types by giving system designers the possibility to scale the platform according to required hardware interfaces and computing power. Figure 2.3 shows an example of a network of components: two Core Processing Modules (CPM), three Remote Data Concentrators (RDC), and two Remote Power Controllers (RPC) linked via two redundant AFDX networks. The CPM components provide the computing power and host the avionics applications, but apart from AFDX they do not provide any I/O hardware interfaces. Instead, the RDC and RPC components provide the required number of sensor/actuator and bus interfaces.

The project also investigates ways of increasing fault tolerance through different reconfiguration capabilities, for example transferring avionics functions from defective modules to other, still operative modules. Finally, the design of a unified tool chain and development environment has led to improvements of the avionics software implementation process.

The research activities and achievements presented in this dissertation are the results of the author's work in the SCARLETT project, in particular in the work packages WP2.4 (Toolset & Simulators Development) and WP4.2 (I/O Intensive Capability Demonstration).

CHAPTER 3

Domain-specific Languages

Domain-specific Languages (DSLs) are programming languages which can be employed to solve problems of a particular application domain. Some well-known examples for DSLs are:

- **SQL** is supported by almost every database system to query, add or delete records.
- **Regular Expressions** are widely used to describe regular languages[1]. Implementations provide a means of matching (recognising) strings of text.
- **VHDL** is a standardised hardware description language employed for integrated circuits design.

What these languages have in common is that they were designed for a single purpose only. In contrast to general-purpose languages like C, Pascal, or Perl, each one of them cannot be used as a replacement for the others. While DSLs are not capable of solving every problem, a good DSL excels at solving problems of its domain.

A DSL employs concepts of its application domain. Its syntax and semantics are based on *domain knowledge*. For example, keywords and operations are named after objects or principles that occur in the problem domain. It also means that, while it is perfectly possible to write syntactically correct programs in general-purpose languages which do not make any sense on a semantic level, the grammar rules of a DSL can forbid such programs because the domain knowledge of what makes sense in a program is available at language design time.

[1]Although many regular expressions libraries provide features that exceed the expressive power of regular languages.

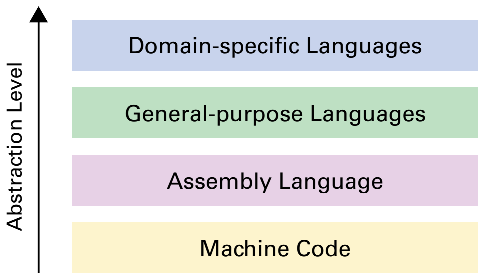


Figure 3.1: Comparison of abstraction levels

Experience shows that working on a higher level of abstraction provides an increase in productivity. Using a DSL means working on a higher level of abstraction compared to working with a general-purpose language (cf. figure 3.1). If the level of abstraction is raised high enough and no concepts of general-purpose programming are left in a DSL, simple knowledge of the domain is sufficient and no general-purpose programming skills are required any more.

However, the introduction of a DSL can only be profitable if the initial effort invested in its development can later be saved due to increased productivity. This means, a DSL can be employed successfully if a well-defined problem domain exists and the problems to be solved require changes to be made frequently to the program, but only infrequently to the DSL or its tools.

DSLs fall into two categories: internal and external DSLs. An internal DSL is embedded into a host language. It can, for example, make use of the host language's control flow features (e. g. loops) and data types. An external DSL, on the other hand, is built from scratch. Anything that is required must be defined anew. While this requires more effort during language design, it makes the language more flexible and expressive.

Further examples and information on DSLs can be found, for example, in [FP10].

3.1 Domain-specific Modelling

The ideas and principles of using DSLs instead of general-purpose languages can be applied to modelling as well: Instead of using a general-purpose modelling language (e. g. UML), a graphical domain-specific language, called a domain-specific modelling language (DSML), can be employed. A DSML uses graphical representations of objects and concepts from its designated problem domain. Therefore, models built from DSMLs can easily be understood, validated, and even created by domain experts without requiring

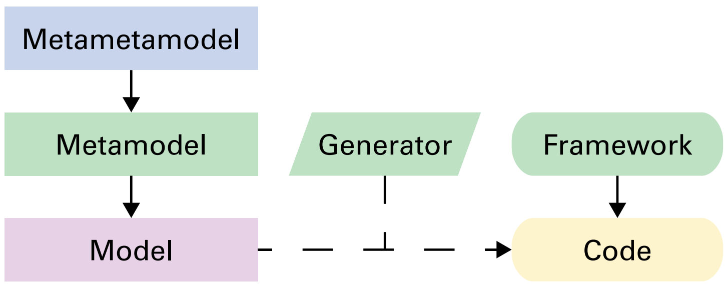


Figure 3.2: DSM Workflow

in-depth knowledge of software engineering principles [KT08]. Adequate tool support makes DSMLs intuitive to use and provides a much shallower learning curve than text-based DSLs (and, of course, general-purpose languages).

Models built from a DSML must adhere to the language syntax. In order to clearly define the syntax of a DSML, the language needs a formal definition. The language definition of a DSML is called a *metamodel*. The language in which a metamodel is defined is called a *meta-metamodel*. Figure 3.2 shows their relationship (cf. [KT08, ch. 4]).

The meta-metamodel is embedded in the DSM tool that is used to design the language. The language designer uses its features to build the metamodel. The metamodel defines all objects, their properties, and relationships that may occur in models of a particular DSML.

When the language designer has completed the metamodel, domain experts can start building models in the new language. They are supported in their task by the DSM tool, as the tool will make sure that only valid models (i. e. conforming to the metamodel) can be built.

Ultimately, the goal of modelling is the generation of code. To that end, apart from the metamodel, the language designer also provides a domain-specific *generator*. It is the generator's task to provide a mapping from the models of the language to the concrete code that can be compiled or interpreted in the target environment. The code generally relies on a domain-specific *framework*, for example a library of functions and some initialisation routines. Well-established principles of operation can be implemented here and therefore do not have to be modelled explicitly in the DSML. The framework is supplied together with the generator by the language designer.

3.2 MetaEdit+

The number of users of a DSML can be potentially very small, depending on the problem domain. It could, for example, be just a dozen employees working with a language used only in-house. It would be rather expensive to develop a complete set of tools specifically for each of those languages. Instead, a number of tools exist that support the design and usage of arbitrary domain-specific modelling languages. Apart from several Eclipse-based solutions, one of them is MetaEdit+ by MetaCase [MC]. MetaEdit+ was chosen for this work due to its ease of use as well as previous positive experiences.

3.2.1 GOPPRR

MetaEdit+ is based on the meta-metamodel "GOPPRR". Like its predecessors OPRR and GOPRR [Kel97, p. 239ff], it was designed specifically for the purpose of metamodelling. The name is an acronym of the elements the meta-metamodel provides. These elements, called metatypes, are used in MetaEdit+'s metamodelling mode to build metamodels, i. e. new DSMLs.

Graphs are individual models, shown as diagrams. They consist of objects, relationships, and roles. Bindings define how objects can be connected via relationships and in which roles.

Objects are the main elements of graphs. An object has a visual representation and a number of properties. The values of its properties can influence the visual representation of an object.

Properties are attributes characterising objects, graphs, roles, or relationships. Their values are of a pre-defined type (text string, integer, enumeration value, etc.)

Ports are optional connection points of an object to which a role can connect. Each port of an object has its own visual representation and position on the object. If an object has no ports, a role can connect to anywhere on the object (provided an appropriate binding rule exists).

Roles connect objects into relationships. Each role can have its own visual representation (e. g. arrow head).

Relationships connect two or more objects via a line and possibly additional visual elements. Each kind of relationship has a specific set of roles, and each role can have its own cardinality.

3.2.2 Metamodelling

MetaEdit+ provides a metamodelling tool for each of GOPPRR's elements. A typical approach would be to start with the Object tool (figure 3.3) and create a new object type for each kind of object, giving them appropriate names and descriptions, then to create properties and add them to the objects. The next step would be to define relationships and roles using the respective tools. Finally, the Graph Tool is used to create graph types. This includes basic properties like a name and graph-level properties as well as the definition of which types of objects, relationships, and roles may appear in the graph. The most important part, however, is the definition of *bindings* [Kel95]. A binding stores the connection information of a relationship (it is not part of the relationship itself). More than one binding can be defined for each type of relationship. A binding B can be defined as a tuple consisting of a relationship type r and a connection set C:

$$
\begin{aligned}
B &= (r, C) \\
C &\subset (L \times R \times P \cup \{\perp\} \times O) \\
L &= \{(l_0, l_1) \mid l_0 \in \mathbb{N},\ \ l_1 \in \mathbb{N}^+ \cup \{\infty\},\ \ l_0 \leq l_1\}
\end{aligned}
$$

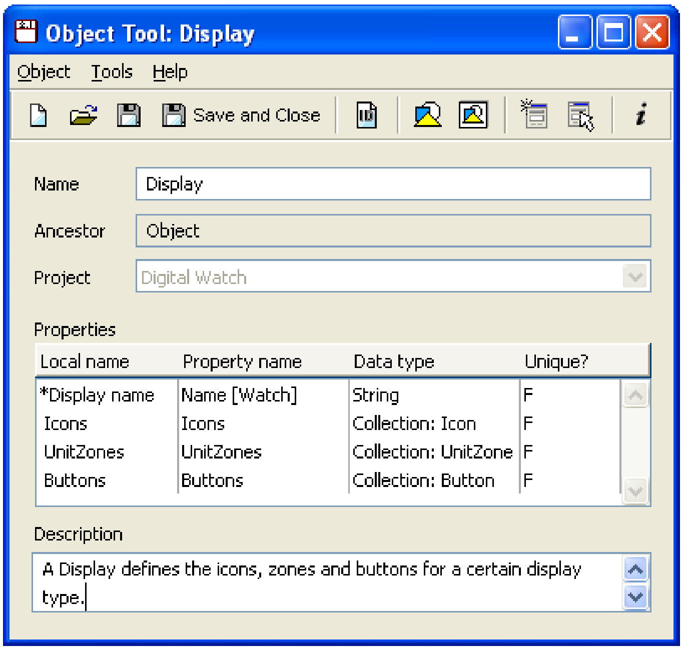
with	r	a relationship type
	l_0	lower bound of cardinality
	l_1	upper bound of cardinality
	R	set of all role types
	P	set of all port types
	$\perp$	the “empty” port
	O	set of all object types

This makes it possible to precisely define constraints regarding the types of objects that can be in relationships with other object types as well as their cardinalities (figure 3.4). The Graph Tool provides two additional features: It is possible to define *nesting* of graphs, i. e. objects in a graph can decompose into graphs of the same or another type. Finally, graph-level constraints can be defined. Connectivity constraints limit the number of roles or relationships an object may be in. Occurrence constraints define a maximum number of occurrences for an object type. Uniqueness constraints require that a specific property is unique amongst all objects of a type in the graph (figure 3.5).

MetaEdit+ also provides a Symbol Editor. This tool is used to design the graphical representation of objects, ports, roles, and relationships. The values of object/port/role/relationship properties can be used as text elements. The visibility of certain parts of a symbol can be made to depend on property values.

3.2.3 Modelling

When the metamodelling is completed, the domain experts can use MetaEdit+ in its modelling mode in order to create models. For each type of graph defined in the metamodel, MetaEdit+ provides a dedicated Diagram Editor (figure 3.6). Only the object and relationship types applicable to the



Name: Display

Ancestor: Object

Project: Digital Watch

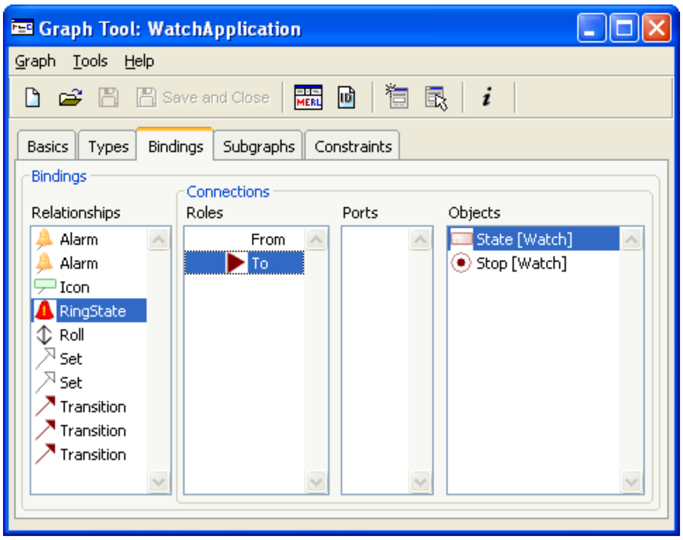
Properties

Local name	Property name	Data type	Unique?
*Display name	Name [Watch]	String	F
Icons	Icons	Collection: Icon	F
UnitZones	UnitZones	Collection: UnitZone	F
Buttons	Buttons	Collection: Button	F

Description

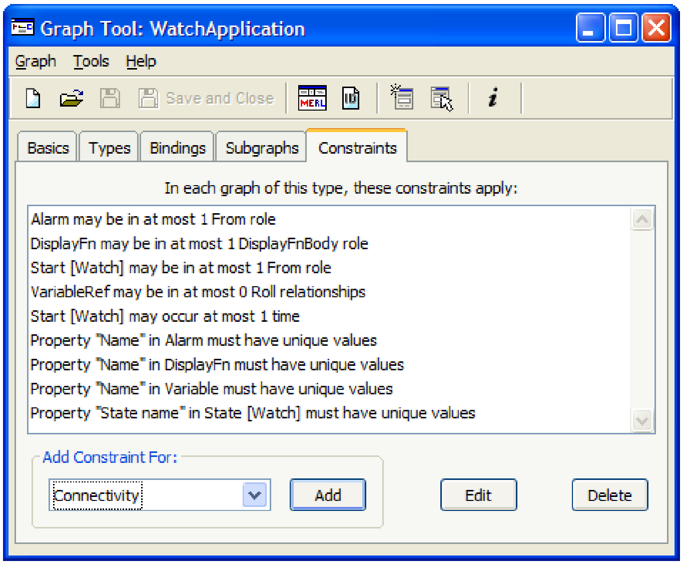
A Display defines the icons, zones and buttons for a certain display type.

Figure 3.3: MetaEdit+ Object Tool



Relationships	Roles	Ports	Objects
Alarm	From		State [Watch]
Alarm	To		Stop [Watch]
Icon			
RingState			
Roll			
Set			
Set			
Transition			
Transition			
Transition			

Figure 3.4: MetaEdit+ Graph Bindings


Graph Tool: WatchApplication
Graph Tools Help
Save and Close
Basics Types Bindings Subgraphs Constraints
Add Constraint For:
Add
Edit
Delete

In each graph of this type, these constraints apply:

Alarm may be in at most 1 From role
DisplayFn may be in at most 1 DisplayFnBody role
Start [Watch] may be in at most 1 From role
VariableRef may be in at most 0 Roll relationships
Start [Watch] may occur at most 1 time
Property "Name" in Alarm must have unique values
Property "Name" in DisplayFn must have unique values
Property "Name" in Variable must have unique values
Property "State name" in State [Watch] must have unique values

Connectivity

Figure 3.5: MetaEdit+ Graph Constraints

particular graph are made available to the modeller. Objects appear in their custom, domain-specific representation. Object properties can be edited via customised dialogue boxes. Objects can only be combined into relationships according to the defined bindings; the editor does not permit invalid connections. Compliance with graph-level constraints is guaranteed as well.

3.2.4 Generators

The last step is the generation of code based on the models the domain experts created. As described before, generators are required for this task. MetaEdit+ supports the generation of code and other outputs via its integrated generator language: MERL (MetaEdit+ Reporting Language). MetaEdit+ provides an editor tool which is used to write MERL generators for code, generator-based text fields in symbols, and object identifiers.

In order to better understand MERL-based generators used in the following parts of this document, a quick overview of MERL syntax will be given here.

MERL was designed to navigate through design models and extract information from model elements. It is stack-based, with the current model element at the top of the stack. When a generator starts, the initial object on the stack is the graph that the generator is given to process. Most generators operate by looping over all or a subset of objects of a graph. This is possible with a `foreach` loop:

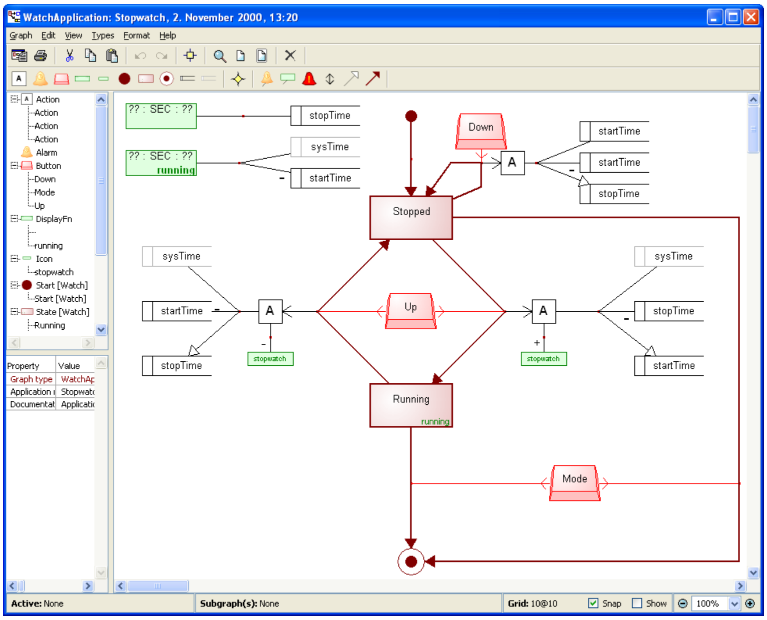


Figure 3.6: MetaEdit+ Diagram Editor

```
foreach .State
{
  :name; newline
}
```

The `foreach` loop starts on the graph level and iterates over all objects of the given type, in this case all objects of type `State`[2]. Inside the loop body, the current `State` event becomes the top element on the object stack and it is possible to access its properties. In this case, the `name` property of each `State` object in the graph is output, followed by a newline.

From the current element it is possible to navigate through bindings to other objects:

```
do ~From>Link~To.State
{
  :name; newline
}
```

Assuming the current object is a `State` (for example by placing this code into the loop body of the first example), the `do` loop iterates through all `From` roles of the current object which are part of a relationship of type `Link` and visits any `State` objects connected via a `To` role to that relationship. These,

[2]Assuming such an object type was defined in the metamodel.

Table 3.1: MERL metatype characters

Character	Metatype
.	Object
:	Property
#	Port
~	Role
>	Relationship

in turn, become the current objects for the body of the `do` loop. In other words, this code example outputs the names of all states that can be reached from the current state.

The strange-looking special characters prefixing the identifiers in the two examples serve a specific purpose: The character defines the kind of metatype that the name refers to (cf. table 3.1).

Loops can be combined with filtering and ordering expressions:

```
foreach .atom where :radioactive = true
              orderby :atomic number;
{
  ...
}
```

This example would loop over all `atom` objects which are radioactive, in order of ascending atomic number. Besides loops, `if-else` statements can be used as well. And, similar to function calls, a generator can also call sub-generators.

It is also important to note that apart from the logical connections between objects and relationships, in MERL it is also possible to obtain information about their positions and layout. You can retrieve an object's x and y coordinates as well as its width and height. Objects that are placed inside the area of the current object can be obtained via the `contents` keyword. Similarly, `containers` retrieves the set of all objects that contain the current object.

Since it is possible to access all information contained in a model, it quickly becomes obvious that MERL provides a very powerful and flexible approach to generating code and other data outputs from GOPPRR-based models. For a complete description of MERL and its syntax, see [MWUG, ch. 5].

CHAPTER 4

Model-based Testing

Model-based testing (MBT) is a process whose purpose is the generation of executable test procedures from models. Instead of writing tests manually, test cases are derived from models somehow related to the SUT [ZSM11b]. In this way MBT provides great potential for test automation. Consequently, as several studies show, the application of model-based testing results in a reduced effort and cost for testing activities [PPW+05, Bin12].

Two different kinds of models can be used as a basis for model-based testing: *System models* represent the desired behaviour of the SUT. They are abstractions of the actual SUT implementation. *Test strategy models*, on the other hand, are models that are created separately from a system model and describe some explicit testing strategy.

Apart from test selection criteria (e. g. structural model coverage, data coverage, requirements coverage or formal test case definitions), [ZSM11b] describes different test generation algorithms:

Random input generation is the simplest method: Random values from the input space of a system are selected and used as inputs. Unfortunately, the time it takes to achieve acceptable model coverage is not predictable.

Graph coverage involves graph search algorithms for finding paths that include all nodes or arcs (e. g. Chinese Postman algorithm) in order to reach sufficient model coverage.

Model checking is used to verify properties of a system. If test cases are specified as reachability properties, a model checker is able to generate traces that eventually reach a given state or transition.

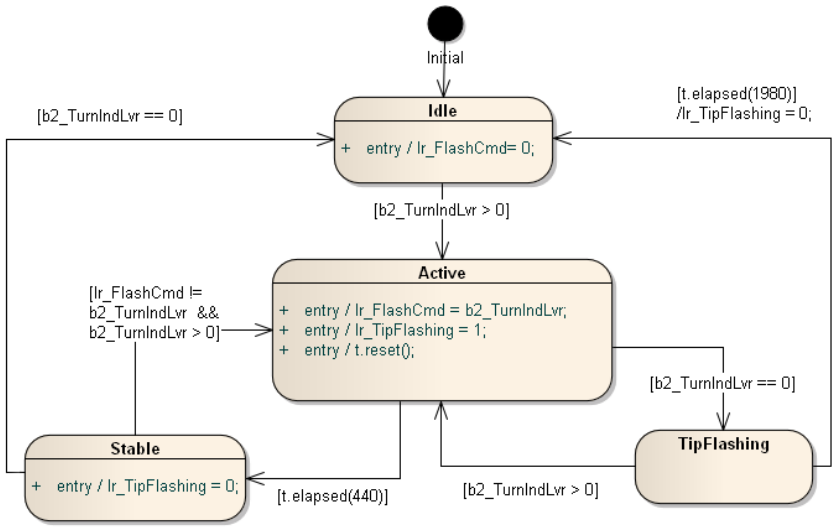


Figure 4.1: Example model: Turn indicator (UML). (Source: `http://mbt-benchmarks.org/`)

Symbolic execution involves the execution of a model (evaluation of expressions) not with actual, but with sets of input values. The result is a symbolic trace that can be instantiated with concrete values. Test cases can be specified as constraints that have to be respected during the symbolic execution.

In model-based testing, two different modes of test execution are possible: *Online testing* involves the evaluation of the model during the actual test execution. Here, the MBT tool interacts directly with the SUT. By contrast, the generation of "conventional" test procedures by an MBT tool and the execution of those procedures in a separate step is called *offline testing*.

As an example, figure 4.1 shows part of a real-world system model of a reactive system, in this case a subset of the functionality of a turn indicator control system is modelled as a UML [OMG11] state machine diagram (a variant of Harel Statecharts [Har87]). A state machine diagram consists of *states* (Idle, Active, Stable, TipFlashing), one of which is the initial state. Each state represents a qualitative aspect of the system's state space. *Extended state variables* capture quantitative aspects of the state space. Their existence avoids an explosion of the number of states in systems larger than any trivial examples. States can have entry and exit actions (`lr_FlashCmd = 0`). States are linked by *transitions*. Each transition can have *guard conditions* (`b2_TurnIndLvr == 0`) which must be satisfied for the transition to be taken, and it can also have *actions* (`lr_TipFlashing = 0`). Guards and actions can be expressions over extended state variables as well as external inputs and outputs of the system. This example models normal direction indication as well as “tip flashing” (lane change indicator).

Test case generation from such a model requires the ability to solve reachability problems [Hui11, p. 30]. In order to travel a path along state machine transitions to reach a given target state, the guard conditions along this path represent constraints that must be solved. The solution is taken as a sequence of inputs to the SUT. The actions defined along the path in the model contain the observable outputs that must be checked by the generated test procedure in order to determine the test verdict.

An important aspect that must be noted is the fact that tests generated from models have the same level of abstraction as the models themselves. As a consequence, implementation details not covered in the model cannot be tested automatically.

For a more exhaustive introduction to model-based testing in the context of embedded systems, refer to [ZSM11a, UL07]. For more information on automated test case generation, see [LP10, Wei10, PVL11, Pel13].

CHAPTER 5

TTCN-3

The Testing and Test Control Notation Version 3 (TTCN-3) is a language designed for writing test specifications. It is based on the Tree and Tabular Combined Notation (TTCN, TTCN-2). While its predecessors were used mostly in telecommunication systems testing, TTCN-3 was developed to be a universal test language. Among its application areas are protocol testing (IPv6, GSM, and UMTS) and its use as a test environment for AUTOSAR [AUT] (a system architecture similar to IMA designed for the automotive domain).

TTCN-3 has been developed and standardised by the Methods for Testing and Specification Technical Committee (TC-MTS) of the European Telecommunications Standards Institute (ETSI). The standard itself is split across several documents, each covering different aspects of TTCN-3. Table 5.1 shows the most important documents. Other documents define alternative representations or extensions to the language. All documents are freely available from ETSI's TTCN-3 website [ETS].

Table 5.1: TTCN-3 Standard documents

Document No.	Description	Reference
ES 201 873-1	Core Language	[ETS09a]
ES 201 873-4	Operational Semantics	[ETS09b]
ES 201 873-5	Runtime Interface	[ETS09c]
ES 201 873-6	Control Interface	[ETS09d]

Table 5.2: TTCN-3 built-in simple types

Name	Description
`boolean`	Truth values (`true`, `false`)
`integer`	Integral values of arbitrary size
`float`	floating point values
`verdicttype`	Test verdicts (`none`, `pass`, `inconc`, `fail`, `error`)

Table 5.3: TTCN-3 built-in string types

Name	Description
`charstring`	ASCII characters
`universal charstring`	Unicode characters
`bitstring`	binary digits (arbitrary number)
`hexstring`	binary digits (multiple of 4 bits)
`octetstring`	binary digits (multiple of 8 bits)

5.1 Language Syntax

Test procedures are written in the TTCN-3 core notation [ETS09a]. This language, although designed specifically for testing, bears many similarities to common programming languages. For example, TTCN-3 provides a strong type system with many built-in types. Table 5.2 shows a list of simple types. New types can be created via aliasing and subtyping, e. g. by constraining an integer type to a range of values [WDT+05, pp. 127–132].

TTCN-3 provides several kinds of string types, for character as well as binary data. Their names are listed in table 5.3. Subtypes of strings can be created by restricting the allowed set of characters and by limiting the string length.

Similar to other languages, it is possible to declare user-defined types. These can be enumerations, records, unions, or sets [WDT+05, pp. 139–150]. While a record is equivalent to a struct type as known from C, a set is similar to them, but the order of its elements does not matter. Another specific feature is that records and sets can have optional elements. An optional element is not required to have an actual value. It is also possible to define lists and arrays of types. While the size of arrays is determined at compile time, the number of elements in a list can change dynamically at runtime. It is, however, possible to define list types with length constraints (similar to string lengths).

There is no implicit type conversion in TTCN-3. If a value of one type shall be used with a value of another type, one of them must be converted explicitly, for example using `int2float` or `float2int`. Variables are declared using the `var` keyword. Declarations can be at any scope level except at the top level. Therefore, global variables are not allowed[1]. Similarly, constants can be defined with the `const` keyword.

A speciality of TTCN-3 is its template system [WDT+05, pp. 173–192]. A template defines one or more values of a specific type. Templates are used when sending and receiving messages. They can define the expected contents of a message, e. g.:

```
type record SomeResponse {
    integer errorcode;
    charstring explanation;
};

template SomeResponse myTmpl := {
    errorcode := (100, 200, 300),
    explanation := ?
};
```

This defines a template `myTmpl` of the record type `SomeResponse`. The template matches any actual records that have an error code of 100, 200, or 300. The question mark implies that any (empty or non-empty) string is accepted in the explanation field.

Functions in TTCN-3 are defined using the `function` keyword. They can have a list of input and output parameters and a return value. A simple example function might look like this:

```
function myAdd(in int i, in float f) return float {
    var float result;
    result := int2float(i) + f;
    return result;
}
```

Functions contain statements as commonly known from C or Pascal. Apart from assignments (`:=`) there are arithmetic (`+`, `-`, `*`, `/`, `mod`, `rem`), relational (`==`, `<`, `>`, `!=`, `>=`, `<=`), logical (`not`, `and`, `or`, `xor`), and string (`&`, `<<`, `>>`, `<@`, `>@`) operators. `for` and `while` loops as well as `if-else` statements can be used just like in C.

One special statement is TTCN-3's `alt` statement [WDT+05, pp. 55–59]. It is used to provide a choice between several (blocking) operations. A typical example would look like this:

```
alt {
    [] port1.receive(someData) {
```

[1]This restriction avoids problems with distributed test components.

```
        doSomething();
    };
    [] port2.receive(otherData) {
        doSomethingElse();
    };
    [] myTimer.timeout {
        setverdict(fail);
    };
}
```

In this case, execution would block until data can be received from ports `port1` or `port2`, or until the timer `myTimer` expires. `someData` as well as `otherData` would be templates (see above) that define the acceptable types and values to be received. In each case, execution would continue in the adjacent block.

As indicated in the previous example, TTCN-3 provides a timer mechanism. Each timer must be declared before use. Afterwards, it can be started with a specified timeout value. Then it can either be stopped (i. e. cancelled), or the timer creates a timeout event. For example, in order to create a delay of two time units, the following code might be used:

```
timer myTimer;

...
myTimer.start(2.0);
myTimer.timeout;
...
```

The `timeout` statement has blocking semantics, therefore it is normally used as part of an `alt` statement.

5.2 Components and Concurrency

In TTCN-3, a test case is a sequence of SUT stimulations and expected results. It is written similar to a function, but instead of a return value it generates a *verdict* (cf. table 5.2). A very simple test case would look like this:

```
testcase MyTC() runs on MyComponent {
    setverdict(pass);
};
```

A test case is declared to run on a particular type of component. A component is an entity that executes TTCN-3 code. Multiple components can exist in parallel, resulting in concurrent execution. Each component has its own local state of variables and timers.

The component executing the test case is called the *Main Test Component* (MTC). This component can create other components, which execute functions declared to run on such component types [WDT+05, pp. 71–76].

A special type of component is the *Test System Interface* (TSI). It does not execute TTCN-3 code, but instead declares the ports that the test system can use to communicate with the SUT.

5.3 Communication

Components can communicate with each other and with the SUT. Communication takes place via ports. The declaration of a component type consists of a list of ports and their types, making up the interface of such a component, e. g.:

```
type component MyComponent {
    port Request pt_req;
    port Response pt_resp;
};
```

This declares a component type that has two ports in its interface: `pt_req` and `pt_resp`. Note that the port itself does not have a direction (i. e. *in* or *out*). Instead, each type of message that can pass through a port type is directed:

```
type port Request message {
    in ReqMsg
};

type port Response message {
    out RespMsg
};

type port ComplexPort message {
    in A;
    out B;
    inout C;
};
```

This last example declares a port type which can receive messages of types `A` and `C` and can send messages of types `B` and `C`. All the port types shown above operate in an asynchronous, message-based manner. Both communication partners use the `send` and `receive` operations to transmit messages.

In client/server architectures communication often happens synchronously in a procedure-based manner. TTCN-3 supports this via procedure-based ports [WDT+05, pp. 87–107]. Instead of messages, a procedure-based port has a set of function signatures:

```
signature login(in charstring user,
                in charstring password)
                return boolean;
signature logout();

type port MyClientPort procedure {
    out login, logout
};

type port MyServerPort procedure {
    in login, logout
};
```

In this example a “client” component can use the `call` operation to perform the login and logout functions and receive their result with the `getreply` operation. The corresponding “server” component would use the `getcall` operation to wait for requests and reply to them with the `reply` operation.

The remaining question is how ports are connected to each other. TTCN-3 distinguishes between *connecting* two ports of different components and *mapping* a port of a component to a port of the TSI, which in turn is expected to be a connection to the SUT [WDT+05, pp. 79–83]. If, for example, the MTC has created two subcomponents, it can connect their ports:

```
testcase TC1() runs on MTC system SUT {
    var Component1 comp1 := Component1.create;
    var Component2 comp2 := Component2.create;

    connect(comp1.port1, comp2.port1);
    connect(comp1.port2, comp2.port2);
}
```

At runtime the TTCN-3 environment checks that all messages which can be sent on one port can also be received on the other port. If not, a test error occurs.

In a similar manner, ports can be mapped for communication with the SUT. The MTC can map its own ports or those of subcomponents to TSI ports:

```
type component MySUT {
    port MsgType MySUTPort
}

type component MyMTC {
    port MsgType MyMTCPort
}
```

```
testcase TC2() runs on MyMTC system MySUT {
    map(self:MyMTCPort, system:MySUTPort);
}
```

In contrast to `connect`, mapping two ports requires that all messages that can be sent on the MTC port can also be sent on the TSI port, as well as that all messages that can be received on the TSI port can also be received on the MTC port. In other words: Ports in the TSI component do not represent ports of the SUT itself, but ports of the whole test environment towards the SUT.

5.4 Runtime Architecture

A TTCN-3 test procedure (or test suite) is *abstract*. The TTCN-3 code itself does not contain information about the concrete format of messages or how they are physically sent to or received from the SUT. Therefore, in order to be executable with a real SUT, a TTCN-3 test procedure needs a suitable runtime environment. It is the runtime environment's task to provide certain services. In particular, its responsibilities include:

- test control functionality (starting, stopping)
- test logging
- mapping between abstract and concrete data types (codecs)
- communication with the SUT

The TTCN-3 standard provides two interfaces that define how a TTCN-3 test system has to provide these services. Figure 5.1 depicts the architecture of a TTCN-3-based test system: The *Test Executable* (TE) is the binary representation of a test procedure written in the TTCN-3 language (cf. section 5.1), produced by a TTCN-3 compiler. The TTCN-3 Control Interface (TCI) defines how tests are started and stopped, how the TE can have data encoded and decoded, and how it can have information logged in the test execution log.

The second interface, the TTCN-3 Runtime Interface (TRI), provides access to two adapters: the SUT adapter and the platform adapter. The purpose of SUT adapter is to provide a mapping between TTCN-3 communication channels and actual hardware (or software) interfaces. This makes it possible for the TE to actually communicate with the SUT. All knowledge about how communication takes place is encapsulated into the SUT adapter, thereby keeping the test procedure abstract. If necessary, the SUT adapter must also reset and activate the System Under Test before test execution.

The platform adapter provides a notion of time on the specific execution platform, in particular via settable timers. It can also provide an interface to arbitrary external functions.

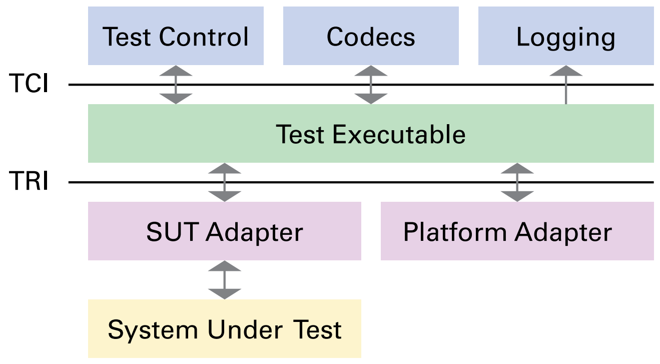


Figure 5.1: TTCN-3 System Architecture

A typical TTCN-3 tool provides test control and logging (usually along with a graphical user interface) as well as a generic platform adapter (e. g. using Windows or POSIX timers). The SUT adapter and any special codecs[2], however, must be provided by the architect or integrator of a concrete test environment.

While this short introduction should give the reader a general overview of TTCN-3 and selected relevant features, a full introduction to TTCN-3, its syntax, concepts, and architecture, including many examples and useful tips, can be found in [WDT+05].

[2]Some primitive codecs may be supplied with the TTCN-3 tool.

PART II

Framework Architecture

CHAPTER 6

Test Case Generation and Execution Process

Before going into the details in the following chapters, this chapter will give an overview of the general workflow and the process of generating and executing test cases in the framework.

An IMA test engineer (domain expert) uses the model-based IMA testing framework in order to perform tests of IMA modules. The process of test case generation and execution is divided into four steps:

1. Step one consists of transforming test requirements into ITML models. This part might have to be repeated if new requirements arise or intended behaviour changes. It requires knowledge about IMA/DME module testing and about modelling in ITML, but not about the details of programming test procedures with different test systems or tools, because ITML provides an abstraction from this.
2. In the second step each model must be processed with the ITML Test Case Generator. This step runs fully automated and must be triggered (from the User Interface) whenever a model has been changed. The test cases will be generated and compiled into executable test procedures.
3. Step three consists of the preparation of the SUT for test execution: Either the actual application software or the Test Agent as well as the required configuration must be compiled and loaded onto the SUT before testing can begin. Manual interaction can be required depending on the module-specific load generation and data loading tools.
4. The fourth step is to execute the compiled test procedures in the test environment on the TTCN-3 test system. This means that the person

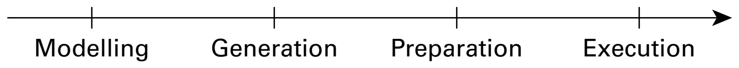


Figure 6.1: Test Case Generation and Execution Process

conducting the tests can select a single test procedure or a batch of multiple test procedures in the User Interface and have them executed. No manual interaction is required during test execution. When the tests have completed, the verdict will be displayed and the test logs will be available for reading or archiving.

The test procedure requires knowledge about the specific configuration of the SUT. Depending on the level of the test (see section 2.4), this will either be a configuration designed specifically for this test, or a pre-defined configuration generated by the module integrator. In the former case, the complete configuration tool chain is required to produce the configuration loads. In those cases where the universal Test Agent is to be used (basically all except Hardware/Software Integration tests), the compiler and load generation tool chain must be available during the testing process.

CHAPTER 7

Test Agent

In order to test the behaviour of an embedded system's operating system, running actual application code[1] is generally inadequate. An application might not make use of all functionality that is supposed to be tested, or it might be very difficult to reach an application state where a desired feature is actually used at runtime.

This problem can be solved by using custom applications specially programmed for testing purposes. These need to make specific API calls depending on the test case that is supposed to be covered by the test. Since the turnaround of compiling, loading, and executing an application on an embedded system is significantly higher than on a desktop computer, it is very desirable to have one universally usable application, instead of one application per test procedure.

Exactly this demand is fulfilled by the universal Test Agent (TA). It has been designed and implemented as an ARINC 653-compatible application that can be loaded into CPM partitions. Each agent instance can execute API calls on behalf of the running test procedure. To that end, the agent offers a remote procedure call interface. This interface requires two AFDX ports—one input port and one output port—per partition. These ports are called *command ports*. They must be chosen (or defined) reasonably. The following criteria apply:

- A maximum message size of at least 8 bytes (recommended: 64 bytes or more)
- A small BAG value for low latency

[1] i. e. application code used upon entry into service

Each partition that a TA instance runs in must have its own dedicated command ports. The remote procedure interface enables the test procedure to trigger three different kinds of calls inside the partition:

API calls directly trigger ARINC 653 API functions.

Scenario calls are complex combinations of API calls, possibly with additional logic or time-critical processing.

Auxiliary calls are helper functions, not resulting in API calls (i. e. without OS-visible side effects).

While the set of API calls is predefined by ARINC 653, scenario calls are custom operations that can be implemented as needed for a specific SUT and test campaign. Scenario call handling is implemented in a C file, which in turn is compiled and linked to the TA binary.

Scenario calls can, for instance, be used to get and set the values of signals. A signal, in this case, is a value that is transmitted as part of an AFDX message, together with other signal values. The module's configuration defines the size, data type and exact position of each signal in the message. Using a scenario call, it is possible to change the value of one signal within an AFDX message without having to transfer the complete message to the test environment and back.

Auxiliary functions do not directly trigger API calls. They provide special functions for the preparation of API and scenario calls, especially for handling buffers. A data table provides a mapping between index numbers and memory for strings and data. Its use avoids the necessity to repeatedly transmit data blocks to be used as parameters for API calls. A definition of the auxiliary functions implemented by the Test Agent can be found in appendix B.

7.1 Test Agent Control Protocol

A test procedure must be able to send different commands to a test agent, while a test agent has to send its responses back to the test procedure. A special protocol has been designed in order to facilitate this communication, the so-called Test Agent Control Protocol (TACP).

Considering the fact that commanding a TA must be possible via ports which were originally defined for different purposes (configured module testing, section 2.4), two important requirements for TACP can be defined:

1. TACP must be usable over sampling ports as well as queuing ports.
2. TACP must be usable over ports that have a maximum message size which is smaller than TACP's maximum PDU size.

Table 7.1: TACP Layer 1

Field	**Size (Octets)**	**Description**
Sequence Number	1	Sequence number of this Layer 1 PDU
Acknowledgement Number	1	Sequence number of last Layer 1 PDU that was received successfully
Flags	1	Layer 1 flags (cf. table 7.2)
Spare	1	Must be set to 0
Payload	*variable*	Contains Layer 2 data (complete or split PDU)

Transmitting protocol-based data over sampling ports is problematic insofar as it is possible to lose frames when writing new data into the output port too fast. New data must only be written into a port if it is known that the other party has received the previous data. Therefore a flow control mechanism is required.

Another problem arises from the fact that, although the underlying IPv4 internetworking layer provides the capability to fragment larger datagrams into smaller units, AFDX forbids the use of fragmentation on sampling ports. Therefore, in order to be able to transmit protocol data units exceeding the maximum frame size, a separate fragmentation/reassembly mechanism must be employed.

TACP defines two protocol layers. TACP layer 1 provides fragmentation/reassembly and flow control, while layer 2 provides the remote procedure call service.

7.1.1 Layer 1

Table 7.1 shows the structure of a TACP layer 1 PDU. Sequence number and acknowledgement number are used for flow control: When a sender intends to transmit new data, it can do so only as soon as it has received a layer 1 frame where the acknowledgement number is equal to the sequence number last transmitted by the sender. When sending the new data, the sender must increase the sequence number by one (modulo 256). The acknowledgement number must always be set to the sequence number last received from the other party. Details on sequence/acknowledgement number initialisation can be found in section 7.2.2.

The *more data* flag specifies if a larger layer 2 PDU has been split into multiple fragments. For all but the last fragment this flag must be set. A

Table 7.2: TACP Layer 1 flags

Name	Value	Description
`L1_MORE_DATA`	`0x01`	Payload is part of a split Layer 2 PDU (but not the last part), more data follows

Table 7.3: TACP Layer 2

Field	Size (Octets)	Description
Partition	1	Source/Target partition number
Process	1	Source/Target process number
Operation ID	2	See Operations table 7.4
Parameters	*variable*	Parameters/results of operation

receiver knows that it has received a full layer 2 frame as soon as it receives a layer 1 frame with the *more data* flag cleared. Layer 1 PDUs carry layer 2 PDUs as their payload. Layer 1 PDUs without payload are used for simple reception acknowledgement.

7.1.2 Layer 2

The layer 2 PDU structure is described in table 7.3. The meaning of its fields depends on the direction in which a PDU is sent: A layer 2 PDU sent to the SUT is called a command PDU. In this case, the partition and process fields address the target that is supposed to carry out the operation specified in the operation field. The rest of the PDU contains operation-dependent parameters.

On the other hand, a layer 2 PDU sent by a Test Agent is called a response PDU. Here, the partition and process fields designate the source which carried out the operation specified in the operation field. In this case, the remainder of the PDU contains the results of the operation.

All fields in a layer 2 PDU are encoded in network byte order (Big Endian). Table 7.4 shows the list of valid operations and their associated values. Each operation has a number of parameters that must be specified in the command PDU, as well as a number of return values that must be sent back in the response PDU. For the definition of required parameters and return values for each operation, refer to [Aer05].

Table 7.4: TACP Operations

Operation	**ID**
API Calls	
`CLEAR_BLACKBOARD`	1
`CREATE_BLACKBOARD`	2
`CREATE_BUFFER`	3
`CREATE_ERROR_HANDLER`	4
`CREATE_EVENT`	5
`CREATE_PROCESS`	6
`CREATE_QUEUING_PORT`	7
`CREATE_SAMPLING_PORT`	8
`CREATE_SEMAPHORE`	9
`DELAYED_START`	10
`DISPLAY_BLACKBOARD`	11
`GET_BLACKBOARD_ID`	12
`GET_BLACKBOARD_STATUS`	13
`GET_BUFFER_ID`	14
`GET_BUFFER_STATUS`	15
`GET_ERROR_STATUS`	16
`GET_EVENT_ID`	17
`GET_EVENT_STATUS`	18
`GET_MY_ID`	19
`GET_PARTITION_STATUS`	20
`GET_PROCESS_ID`	21
`GET_PROCESS_STATUS`	22
`GET_QUEUING_PORT_ID`	23
`GET_QUEUING_PORT_STATUS`	24

Table 7.4: TACP Operations (continued)

Operation	ID
GET_SAMPLING_PORT_ID	25
GET_SAMPLING_PORT_STATUS	26
GET_SEMAPHORE_ID	27
GET_SEMAPHORE_STATUS	28
GET_TIME	29
LOCK_PREEMPTION	30
PERIODIC_WAIT	31
RAISE_APPLICATION_ERROR	32
READ_BLACKBOARD	33
READ_SAMPLING_MESSAGE	34
RECEIVE_BUFFER	35
RECEIVE_QUEUING_MESSAGE	36
REPLENISH	37
REPORT_APPLICATION_MESSAGE	38
RESET_EVENT	39
RESUME	40
SEND_BUFFER	41
SEND_QUEUING_MESSAGE	42
SET_EVENT	43
SET_PARTITION_MODE	44
SET_PRIORITY	45
SIGNAL_SEMAPHORE	46
START	47
STOP	48
STOP_SELF	49

Table 7.4: TACP Operations (continued)

Operation	ID
SUSPEND	50
SUSPEND_SELF	51
TIMED_WAIT	52
UNLOCK_PREEMPTION	53
WAIT_EVENT	54
WAIT_SEMAPHORE	55
WRITE_SAMPLING_MESSAGE	56
Auxiliary Functions	
AUX_CREATE_DATA_TABLE	1000
AUX_GET_DATA_TABLE_ENTRY	1001
AUX_SET_DATA_TABLE_ENTRY	1002
AUX_RESERVE_DATA_TABLE_ENTRY	1003
AUX_CLEAR_DATA_TABLE_ENTRY	1004
AUX_DELETE_DATA_TABLE	1005
Scenario Calls	
SCE_OPEN_SIGNAL_PORTS	2000
SCE_READ_SIGNAL	2001
SCE_WRITE_SIGNAL	2002

The encoding of parameters and results is very similar to XDR [SM87]. In particular, the actual encoding rules for parameters and return values are as follows:

1. Parameters are encoded in the order in which they are listed in the standard, i.e. the leftmost parameter first. OUT parameters that are not numeric or enumeration types are transmitted as an index number into the data table.

2. Return values are encoded in the following order: Any OUT parameters in the order in which they are listed in the standard, followed by

the return code. Invalid OUT parameters are transmitted as value 0. OUT parameters that are neither numeric nor enumeration types are not transmitted in the response, as their values can be obtained from the data table.

3. Return codes (`RETURN_CODE_TYPE`) and other enumeration types are encoded as 4-byte integers in network byte order.
4. 8-bit, 16-bit, and 32-bit numeric types are encoded as 4-byte integers in network byte order.
5. 64-bit numeric values are encoded as 8-byte integers in network byte order.
6. The members of a record are encoded in the order in which they appear in the standard, i. e. topmost member first.
7. Character strings and data buffers are not transmitted directly. Instead, an index number into the data table is transmitted as a 4-byte integer in network byte order.
8. Exception: The data buffer parameters to auxiliary functions for data table handling are transmitted in binary coding as an array of characters. The array is preceded by its 4-byte length parameter (in network byte order). The end of the buffer is padded to a 4-byte boundary. The size of the padding is not included in the value of the length parameter.

7.1.3 Coding Example

As an example, the coding of an API call command and its response are shown here:

Command

Destination: Partition 10, Process 100

Operation: `CREATE_SAMPLING_PORT`

Parameters:

`SAMPLING_PORT_NAME:`	`”EXAMPLE_DATA_1”`
`MAX_MESSAGE_SIZE:`	`64`
`PORT_DIRECTION:`	`DESTINATION`
`REFRESH_PERIOD:`	`500`

Response

Source: Partition 10, Process 100

Operation: `CREATE_SAMPLING_PORT`

Parameters:

```
SAMPLING_PORT_ID:   23
RETURN_CODE:        NO_ERROR
```

Note: We assume that the zero-terminated string "`EXAMPLE_DATA_1`" has been placed in entry 3 of the data table beforehand.

Coding of command PDU on protocol layer 2:
```
0A 64 00 08 00 00 00 03 00 00 00 40 00 00 00 01 00 00 01 F4
```

Coding of response PDU on protocol layer 2:
```
0A 64 00 08 00 00 00 17 00 00 00 00
```

Coding of command and response, including acknowledgements, on protocol layer 1 (command split into two layer 1 PDUs, response sent as one layer 1 PDU):
```
→ 01 00 01 00 0A 64 00 08 00 00 00 03 00 00
← 00 01 00 00
→ 02 00 00 00 00 40 00 00 00 01 00 00 01 F4
← 01 02 00 00 0A 64 00 08 00 00 00 17 00 00 00 00
→ 02 01 00 00
```

7.2 Runtime Behaviour

7.2.1 Initialisation

During module start-up, all loaded Test Agent instances initialise themselves in their partition. Initially, all partitions run in INIT mode, executing an initialisation process (cf. section 2.2.3). The entry point for this process is the TA initialisation routine (`ta_init`).

While running in INIT mode, an application must allocate the resources it will need during normal operation. The first resources that the TA needs are its command ports. In order to open the ports, the TA needs to know the port parameters (name, maximum message size, direction, and refresh rate). The correct parameters must be passed to the `CREATE_SAMPLING_PORT` API call, otherwise port creation will fail. Since the TA cannot be told the correct parameters at runtime (as communication is not possible yet), it must know those beforehand. This is accomplished by the TA configuration parser. This parser generates a table (`ta_ports`) of all ports and their parameters from the actual module configuration for a partition. This table is linked to the TA binary and loaded into the module partition with it. At runtime, the initialisation process accesses this table to open the partition's command ports.

Next, the TA creates and initialises an array to store information about the processes in the partition. The init process is always placed at array index

0. Each process has its own buffer that stores commands addressed to the process after they have been received via the partition's input command port. Such a buffer is created for the init process and referenced in the process array. Then, the TA creates another buffer used as a global output queue. All processes of the partition place their response messages in this buffer in order to have them sent via the partition's output command port.

7.2.2 Sequence Number Initialisation

The sequence number handling of TACP layer 1 requires a special set-up procedure. When the TA starts up, the test environment might already be running and its sequence and acknowledgement numbers can have arbitrary values. If the TA simply initialised its sequence and acknowledgement numbers to 0, the first frame it received from the other party would likely have a different sequence number, and the frame's contents would be interpreted as a new command, although it was in fact an old one[2].

In order to avoid this situation, there is a delay of up to one second before the TA starts transmitting. During this time it waits for an incoming frame on its command port. If one is received, the TA's sequence number is initialised with the received acknowledgement number, and the TA's acknowledgement number is initialised with the received sequence number. If no frame is received during that second, both numbers are set to 0.

7.2.3 Main Loop

When all initialisation has been completed, the init process starts the execution of the TA's main loop (`ta_main_loop`). It does *not* set the partition into normal operation mode, but instead leaves it in INIT mode. The reason for this is that if the partition were switched into normal mode, it would not be possible to command the TA to allocate more resources, open additional ports, or perform any INIT mode testing. Therefore, the transition to normal mode is only performed when the TA executes the corresponding API call as commanded by the Test Environment.

Any running processes in the partition execute the same main loop. It consists of the following steps:

1. Try to receive data from the partition's input command port. On success, call the TACP protocol stack with the received data.

2. Check the process' own command input buffer. If it is not empty, retrieve the next element and call the command handler.

3. If data can be sent, check the global send queue. If it is not empty, retrieve the next element and send it via the partition's output command port.

[2]Recall that sampling port data is always retransmitted periodically.

After each main loop cycle, a periodic process will call `PERIODIC_WAIT`, thus suspending itself until its next activation period.

7.2.4 Protocol Handling

The Test Agent's TACP protocol implementation handles the sending and receiving of TACP messages. Whenever a process receives a new frame from the TACP command port, it calls the layer 1 receive function (`recv_tacp_l1`). This function stores the received acknowledge number and then checks if the sequence number differs from the sequence number of the previously received frame. If so, the frame's contents are new, and the function appends the payload of the frame to a static buffer. Then it checks the frame's flags field. If the *more data* flag (`L1_MORE_DATA`) is set, the layer 2 PDU is not yet complete, and so the function returns. But if the flag is not set, the contents of the static buffer now constitute a complete layer 2 frame. This frame is handed to the layer 2 receiving function (`recv_tacp_l2`).

The layer 2 receiving function's task is very simple: It must route the layer 2 frame to the correct process command input buffer. To do that, it checks the layer 2 header's address fields and finds the matching process buffer in the process information array.

When a process finds a new command in its command input buffer (while performing the second step of the main loop), it calls the command handler function (`handle_tacp_cmd`). This function examines the command's operation field. Depending on the numerical range the operation code is in, the function calls either the API call handler, the scenario handler, or the auxiliary function handler. These three handlers are very similar. Each has a jump table referencing the specific handler for each operation code. After performing a range check, the handler directly calls the function referenced by the operation code.

Each specific handler now has to check that the number of parameters supplied with the command matches the particular call. If so, it can extract the parameter values from the command buffer and make the actual call. Afterwards, it places the return code and any output parameters in the command buffer. Then it passes the buffer to the TACP layer 2 send function (`send_tacp_l2`), thereby turning the command into a response.

The layer 2 send function accepts a complete layer 2 response message. It must ensure that the layer 1 frames it generates are small enough to fit through the command output port (i. e. frame lengths must be smaller or equal to the maximum message size of the port). So the function generates a layer 1 frame in a local buffer, initialises the header fields (sequence and acknowledge numbers are initialised with 0 as their actual values cannot yet be determined), and fills it with as much data of the layer 2 frame as possible. If not all of the layer 2 frame fits, it sets the *more data* flag (`L1_MORE_DATA`) in the layer 1 header. Then it calls the layer 1 send function (`send_tacp_l1`).

The layer 2 send function repeats these steps until the complete layer 2 message has been fragmented into layer 1 frames.

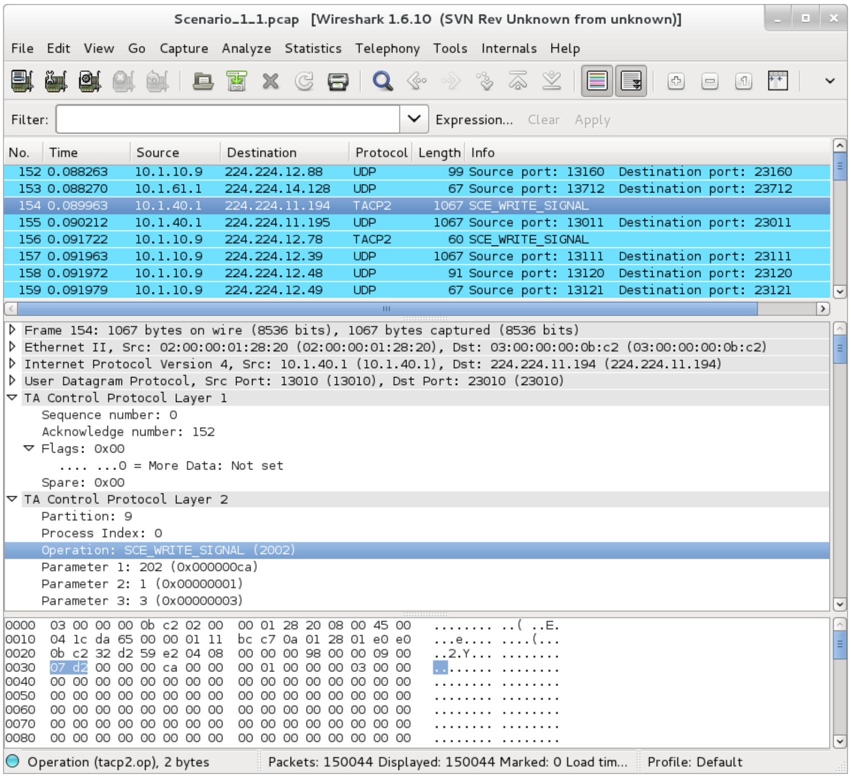
The layer 1 send function simply places each frame that is passed to it into the global send queue of the partition and returns. As soon as a process executes the last step of the main loop, it checks if the last received acknowledgement number is equal to the last sent sequence number. If so, new data can be sent without the risk of losing a frame. In this case, the process tries to obtain a frame from the global send queue. If the queue is empty, no new data needs to be sent right now, and the last frame is simply repeated (possibly with an updated acknowledgement number). If a frame was retrieved from the queue, its sequence number is set to the sequence number of the last frame plus 1 (modulo 256), and its acknowledgement number is set to the last received sequence number. Then the new frame is sent via the command output port.

7.3 Wireshark TACP Dissector

In general, during the integration phase of a newly developed system or software, it is necessary to find problems pertaining to the communication between the system and its (test) environment. This becomes even more difficult with complex high-speed communication networks. A common method of debugging is the use of a network analyser, the open-source tool *Wireshark*[3] being a popular choice. The framework provides a plug-in that enables Wireshark to understand and dissect TACP frames (cf. figure 7.1). This can be an invaluable aid for the test engineer/integrator when analysing communication problems with a Test Agent.

In particular, the plug-in facilitates two things: First, it marks TACP traffic in the overview of captured frames, making it easily recognisable. Second, the plug-in is able to analyse frames of both TACP protocol layers. It displays and interprets the values of all header fields. The value of the operation field on layer 2 is decoded into a human-readable form according to table 7.4, and the numeric values of the operation's parameters are displayed as well.

[3]`http://www.wireshark.org`


Scenario_1_1.pcap [Wireshark 1.6.10 (SVN Rev Unknown from unknown)]

File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help
Filter:
Expression... Clear Apply

```
No.  Time      Source      Destination     Protocol Length Info
 152 0.088263  10.1.10.9   224.224.12.88   UDP        99 Source port: 13160  Destination port: 23160
 153 0.088270  10.1.61.1   224.224.14.128  UDP        67 Source port: 13712  Destination port: 23712
 154 0.089963  10.1.40.1   224.224.11.194  TACP2    1067 SCE_WRITE_SIGNAL
 155 0.090212  10.1.40.1   224.224.11.195  UDP      1067 Source port: 13011  Destination port: 23011
 156 0.091722  10.1.10.9   224.224.12.78   TACP2      60 SCE_WRITE_SIGNAL
 157 0.091963  10.1.10.9   224.224.12.39   UDP      1067 Source port: 13111  Destination port: 23111
 158 0.091972  10.1.10.9   224.224.12.48   UDP        91 Source port: 13120  Destination port: 23120
 159 0.091979  10.1.10.9   224.224.12.49   UDP        67 Source port: 13121  Destination port: 23121
```

```
▷ Frame 154: 1067 bytes on wire (8536 bits), 1067 bytes captured (8536 bits)
▷ Ethernet II, Src: 02:00:00:01:28:20 (02:00:00:01:28:20), Dst: 03:00:00:00:0b:c2 (03:00:00:00:0b:c2)
▷ Internet Protocol Version 4, Src: 10.1.40.1 (10.1.40.1), Dst: 224.224.11.194 (224.224.11.194)
▷ User Datagram Protocol, Src Port: 13010 (13010), Dst Port: 23010 (23010)
▽ TA Control Protocol Layer 1
    Sequence number: 0
    Acknowledge number: 152
  ▽ Flags: 0x00
      .... ...0 = More Data: Not set
    Spare: 0x00
▽ TA Control Protocol Layer 2
    Partition: 9
    Process Index: 0
    Operation: SCE_WRITE_SIGNAL (2002)
    Parameter 1: 202 (0x000000ca)
    Parameter 2: 1 (0x00000001)
    Parameter 3: 3 (0x00000003)
```

```
0000  03 00 00 00 0b c2 02 00  00 01 28 20 08 00 45 00   ........ ..( ..E.
0010  04 1c da 65 00 00 01 11  bc c7 0a 01 28 01 e0 e0   ...e.... ....(...
0020  0b c2 32 d2 59 e2 04 08  00 00 00 98 00 00 09 00   ..2.Y... ........
0030  07 d2 00 00 00 ca 00 00  00 01 00 00 00 03 00 00   ........ ........
0040  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ........ ........
0050  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ........ ........
0060  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ........ ........
0070  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ........ ........
0080  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ........ ........
```

Operation (tacp2.op), 2 bytes | Packets: 150044 Displayed: 150044 Marked: 0 Load tim... | Profile: Default

Figure 7.1: TACP protocol dissector for Wireshark

CHAPTER 8

TTCN-3 Environment

While chapter 5 provided a general overview of TTCN-3, this chapter contains a discussion of a specific test environment used in the ITML framework. During the course of the SCARLETT research project, *TTworkbench* from Testing Technologies [TTW] was selected as a TTCN-3-based test execution tool.

In the ITML framework, the test environment must serve two particular purposes:

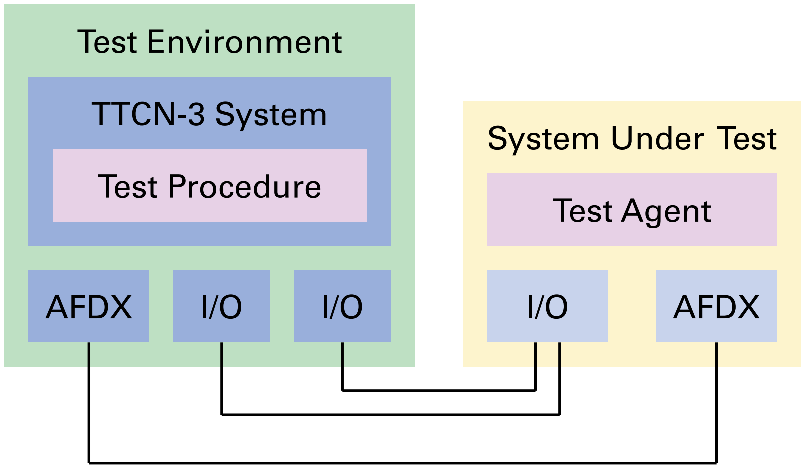


Figure 8.1: Test Environment and System Under Test

- The test environment must provide a way of communication with the Test Agent(s) residing in the SUT. The test procedure must be able to command the Agents and observe their responses.
- The test environment must interface with all input/output hardware of the System Under Test. It must provide the test procedure with a means to stimulate all types of inputs and to observe all types of outputs of the SUT.

A test environment may consist of several different hardware components. Each of the components may be responsible for a different kind of I/O hardware interface, e. g. discrete input/output, analogue input/output, torque sensors, temperature sensors, as well as AFDX, CAN, or ARINC 429 bus interfaces.

In the TTCN-3 test system architecture (cf. figure 5.1 on page 60), the SUT Adapter (SA) constitutes the interface between the test procedure and the SUT. Therefore, the SA must translate all TTCN-3 communication taking place over TSI ports into actual hardware I/O.

8.1 Proxy SA

Implementing an SA for a single hardware interface is rather straightforward. But, since there can exist only one SA in a test system, there are two alternative ways of implementing the SA in case of multiple separate hardware interfaces:

- A single, monolithic SUT Adapter implementing access to all hardware interfaces, or
- a Proxy SUT Adapter providing a plug-in facility for individual sub-SAs (cf. [WDT+05, p. 211]).

Choosing the latter alternative provides some advantages. The SA becomes more modular and can easily be adjusted to the actual test interface hardware. While the necessary adjustments when changing a hardware interface require source code changes and recompiling of the monolithic SA, the Proxy SA requires no changes at all, and only a different sub-SA needs to be plugged into the architecture.

Each sub-SA implements access to one specific kind of hardware. It is implemented exactly like a single SA, and it does not need to be aware of the presence of other sub-SAs.

The implementation of the Proxy SA itself is very simple. The proxy has to implement the functions required by the TRI interface for an SA. At runtime, it has to forward function calls to the registered sub-SAs. According to [ETS09c, p. 18ff], the following functions are required:

- `triSAReset`

- `triExecuteTestCase`
- `triMap`
- `triUnmap`
- `triEndTestCase`
- `triSend`[1]
- `triCall`[1]
- `triReply`[1]
- `triRaise`[1]

The Proxy SA keeps a list of all registered sub-SAs (`subsaList`). The sub-SAs, like the Proxy SA itself, are derived from class `TestAdapter`. Calls to `triSAReset`, `triExecuteTestCase`, and `triEndTestCase` are passed to all sub-SAs.

Calls to `triMap` are passed to all sub-SAs in turn, until one sub-SA indicates success of mapping the port to its hardware interface (i. e. the sub-SA responsible for the port to be mapped has been found). The Proxy SA associates the port with this sub-SA in a hash map (`portMap`).

When one of the functions `triSend`, `triCall`, `triReply`, `triRaise`, or `triUnmap` is called, the Proxy SA can simply look up the sub-SA responsible for the particular port in its hash map. Then it forwards the call to that sub-SA exclusively.

The reference implementation of the Proxy SA for the ITML framework can be found in appendix C.1.

8.2 TACP SA

The TACP SA is one of the sub-SAs registered in the Proxy SA. Its purpose is to provide access to two codecs[2] which are required in the framework: the TACP codec and the FLOAT codec.

The TACP codec encodes and decodes messages sent via the command ports to and from the Test Agents. On the TTCN-3 level, a TACP layer 1 PDU is defined as follows[3]:

```
type record tacp_l1_pdu {
    uint8 seq_no,
    uint8 ack_no,
```

[1]These functions also have multicast and broadcast variants, which must be implemented as well if such communication methods are used.

[2]Codecs are not part of the TRI interface, but they are retrieved via the `getCodec` function of the SUT Adapter.

[3]The following type definitions can also be found in appendix C.5.

```
    uint8 flags,
    uint8 spare,
    tacp_l2_pdu l2 optional
} with {
    encode ”TACP_CODEC”
};
```

The record contains all header fields of a TACP layer 1 PDU. The `l2` payload field is marked as optional, since a layer 1 PDU does not necessarily contain any payload. The `with encode` clause explicitly tells the TTCN-3 runtime system which codec to employ in order to encode instances of this type.

Consequently, a TACP layer 2 PDU is defined in the following way:

```
type record tacp_l2_pdu {
    uint8 partition,
    uint8 process_idx,
    Operation operation,
    ParameterList parameters
};
```

The `tacp_l2_pdu` record contains the addressing fields of a layer 2 PDU, an operation ID enumeration field, and a list of parameters (or return values) for the operation. This list type is simply defined as:

```
type record length (0..64) of uint32 ParameterList;
```

The TACP codec is able to convert instances of type `tacp_l1_pdu` into their correct binary representation and vice versa (cf. section 7.1). The codec's `encode` and `decode` functions are automatically called by the TTCN-3 runtime system when messages of type `tacp_l1_pdu` are passed through a TSI port.

On this level, all parameters and return values of operations are treated as 4-byte integers. In order to interpret an integer parameter as a float value, it can be converted using the FLOAT codec. A simple TTCN-3 helper function is provided for this purpose:

```
function intAsFloat(in uint32 i) return float32 {
    var float32 f;
    var bitstring bs := int2bit(i, 32);
    var integer r := decvalue(bs, f);
    return f;
}
```

As no port communication is taking place here, the codec is called directly through the `decvalue` function, which decodes a value from its binary representation. The FLOAT codec is employed automatically, because the `float32` type is defined as follows:

```
type float float32 with { encode ”FLOAT_CODEC” };
```

8.3 TACP Protocol Handling

The actual TACP protocol stack implementation on the test environment side resides in a TTCN-3 component. This component, called `TACPHandler`, is responsible for handling communication with one Test Agent instance. If a specific test setup utilises multiple TAs, each will require its own protocol handler instance.

The `TACPHandler` component is defined to have three ports. Two ports, `tacpToCPM` and `tacpFromCPM`, must be mapped to AFDX TSI ports. The AFDX network must be configured so that these AFDX ports are connected to the command input and output ports of the respective Test Agent instance. Messages sent and received through these ports are encoded and decoded by the TACP codec (cf. section 8.2). The third port, named `cmd`, permits only procedure-based communication. The test procedure connects to this port and uses the TTCN-3 `call` statement to trigger API or scenario calls according to the signatures defined for the individual commands.

Each handler component instance executes the function `runHandler`. This function consists of an infinite loop. Inside this loop two `alt` statements are employed to allow the reception of procedure-based calls via the `cmd` port as well as responses from the TA via the `tacpFromCPM` port.

Whenever a call is made over the `cmd` port, the `apicall` alt statement determines which command is requested and builds a suitable command PDU. Depending on the operation, the required parameters, passed along via the procedure-based call, are added to the PDU as well. The PDU is then placed into a ring buffer which serves as a send queue. Now, according to the flow control rules of TACP layer 1, the handler checks if it can send new data: If the last received acknowledgement number is equal to the last sent sequence number and the send queue is not empty, the next PDU is retrieved from the send queue, and its sequence and acknowledgement numbers are set. Finally, the handler sends the PDU via the `tacpToCPM` port to its corresponding TA instance.

The second alt step, `response`, receives PDUs from the TA instance via the `tacpFromCPM` port. Whenever a frame arrives, its sequence number is checked to determine if the PDU contains new data. If there is a layer 2 payload, its operation and parameters are checked for consistency, and then a matching reply is sent back over the `cmd` port to fulfil the command that was requested before.

The full reference implementation of the TACP protocol handler component for the ITML framework can be found in appendix C.6.

8.4 Test Procedures

The aforementioned components and features serve as the foundation for the actual test procedures. These test procedures are themselves implemented as TTCN-3 modules, which have been generated from ITML models. The next chapter will provide an in-depth discussion of ITML modelling.

CHAPTER 9

IMA Test Modelling Language

This chapter provides extensive insight into the domain-specific language that has been developed for the ITML framework.

As explained in chapter 3, DSLs are designed for specific purposes. The goal behind ITML modelling is to enable the user (domain expert) to describe test cases, and to automatically generate executable test procedures from modelled behaviour.

9.1 Domain Analysis

The first step in the development of a domain-specific language is the analysis of its intended usage domain. The analysis shall uncover the relevant entities and concepts, their relationships, and rules governing the domain. Chapter 2 has provided a general overview of the IMA domain. Several rules shall be kept in mind during the analysis (cf. [Wil04]):

1. Find the right level of abstraction. Allow expressiveness within the domain, but without being too general.
2. Avoid concepts that seem good from a programming point of view, but that do not have a counterpart in the domain (and which domain experts might not understand).
3. Keep the language as small as possible to reduce complexity.
4. If required, prefer several languages for different aspects of the domain instead of one big language which covers everything.

Refer to [Mew10, ch. 2.3] for a more general discussion of language design.

Moreover, as discussed briefly in section 2.4, there are three different scopes of IMA testing. Consequently, all of them will be examined, one by one.

9.1.1 Bare Module Testing: Configuration

As the name suggests, Bare Module testing involves only the *bare* module hardware and its operating system. There is no predefined configuration for the module. Instead, it is the test engineer's task to supply an adequate configuration for each test case. Therefore, the module configuration is part of the ITML language domain for this testing scope. An overview of module configuration has been given in section 2.3. It lists the relevant characteristics that must be respected for the analysis.

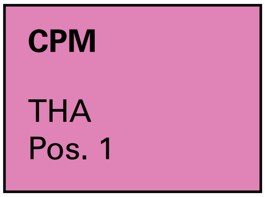
A module configuration specifies properties of the module on global and partition levels. For example, suppose we would like to have a module of type "CPM" with its pin programming set to position 1. We would like to configure two partitions. Each partition has a specified amount of RAM as well as its own scheduling characteristics. Each partition also has its own inputs and outputs. Since CPM modules only support AFDX and RAM communication, only ports of those types can be configured[1]. Ports have a direction, a port characteristic ("sampling" or "queuing") and a maximum message size.

We can now compile a list of all required concepts for the configuration domain together with their properties, relationships, constraining rules, and representation. The chosen representations are of a rather abstract nature, as the concepts, apart from the module itself, have no physical form. These representations will later be used in the DSL.

[1]Additional types of I/O interfaces would be required in order to support other module types.

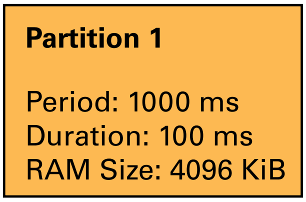
Module

The module represents the SUT as a whole. It has configuration parameters affecting the module in its entirety.

Properties	The name of the module. The module's location (i. e. pin programming).
Relationships	Connected to partitions.
Rules	The module name is unique within the configuration. The module location is unique within the configuration.
Representation	

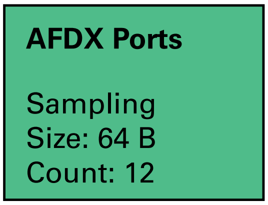
Partition

The partition represents an ARINC 653 partition inside a module. It has partition-related configuration and links to other concepts which exist for each partition.

Properties	The name of the partition. The scheduling period of the partition. The slice duration of the partition. The amount of RAM reserved for the partition.
Relationships	Connected to module. Connected to AFDX ports. Connected to RAM ports.
Rules	Cannot be connected to more than one module. The partition name is unique within the configuration. The slice duration must be less than the scheduling period.
Representation	 Partition 1 Period: 1000 ms Duration: 100 ms RAM Size: 4096 KiB

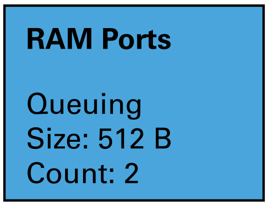
AFDX Ports

The AFDX Ports concept represents a group of AFDX ports with similar properties. Note that the individual ports do not have names assigned to them here, since these are not relevant.

Properties	The port characteristic (sampling/queuing). The maximum message size. The actual number of ports.
Relationships	Connected to a partition, either as inputs or outputs.
Rules	Cannot be connected to more than one partition in case of output ports.
Representation	

RAM Ports

The RAM Ports concept represents a group of RAM ports with similar properties. Note that the individual ports do not have names assigned to them here, since these are not relevant.

Properties	The port characteristic (sampling/queuing). The maximum message size. The actual number of ports.
Relationships	Connected to two partitions, to one as inputs and to one as outputs.
Rules	Cannot be connected to more than two partitions. Cannot be input ports for more than one partition. Cannot be output ports for more than one partition.
Representation	

9.1.2 Bare Module Testing: Behaviour

Apart from the (static) configuration part, bare module testing requires a specification of the (dynamic) behaviour. Bare module testing is concerned with the module's operating system and the API and resources it provides. Therefore, in order to find the relevant concepts in this domain, we need to refer to section 2.2 and [Aer05].

Since testing takes place by having the universal test agent (cf. chapter 7) run inside the partition, we need partitions and processes. Inside a process the possible actions, i. e. API calls, as well as their sequence have to be specified. A simple way of achieving this is by using flowcharts. They require start and end points as well as connecting arrows. A node can have multiple incoming connections, and one or more outgoing connections. If there is more than one outgoing connection, the connections must have different conditions assigned to them, as the behaviour is required to be deterministic.

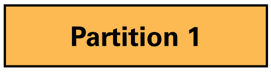
Some ARINC 653 resources are defined and created at runtime (contrary to declaring them in the module configuration). This involves buffers, blackboards, and semaphores. Ports, however, come from the module configuration. Only ports which have been previously declared in the module configuration can be opened at runtime. We need a facility which provides access to them, without having to re-declare what is in the configuration.

In order to work properly with ports, two other concepts are required: First, we need messages which we can send through a port. And second, in order to check correct reception, we need a way to reference the opposite endpoint of a communications channel.

This information allows us to give a complete list of the concepts, their properties, relationships, constraining rules, and representations required for behaviour modelling.

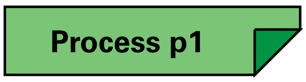
Partition

The partition defines the context in which actions are to take place, i. e. an ARINC 653 partition containing a Test Agent which is to be stimulated.

Properties	The partition name.
Relationships	Connected to process.
Rules	None.
Representation	

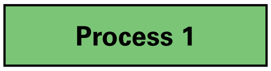
Process Configuration

ARINC 653 processes are created at runtime. They have process-related configuration properties.

Properties	The name of the process. The scheduling period of the process. The scheduling priority of the process. The stack size reserved for the process.
Relationships	Connected to related API calls.
Rules	None.
Representation	

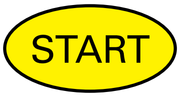
Process

The process defines the context in which actions are to take place inside a partition, i. e. a process of a Test Agent which is to execute API calls.

Properties	The process name.
Relationships	Connected to API call.
Rules	None.
Representation	

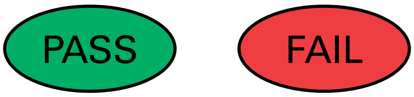
Test Start

A control flow requires a distinguished start location. This is represented by the start symbol.

Properties	None.
Relationships	Connected to partition.
Rules	Must occur exactly once per graph.
Representation	

Test Pass, Test Fail

It must be possible to set the verdict of the test procedure. This is achieved by using the pass and fail symbols within the diagram.

Properties	None.
Relationships	Incoming connections from API calls.
Rules	None.
Representation	

API Call

The central elements of a behaviour specification are ARINC 653 API calls. They shall be executed by the Test Agent running in a partition on the module. There are numerous different API calls, and for brevity not all will be listed here.

Properties	API call parameters, depending on type of API call.
Relationships	Connections to and from other API calls, processes, or complements. Connections to parameter objects, depending on type of API call.
Rules	Depending on API call, requires the correct number and type of connected parameters.
Representation	Create Blackboard

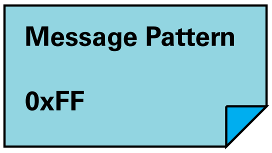
Complement

During I/O testing it is necessary to check for correct reception or to send a message through the opposite endpoint of a port. The complement operation provides a means to do this in a generic manner. This means, for example, that a *Write complement* operation on an AFDX input port will make the test environment send a message so that the partition can receive it from the respective AFDX input port with a *Read_Sampling_Port/Receive_Queuing_Port* API call.

Properties	The operation to be performed (read or write).
Relationships	Connections to and from other API calls, processes, or complements. Connections to port and message pattern parameter objects.
Rules	Must be connected to exactly one message pattern. Must be connected to exactly one port.
Representation	Read complement

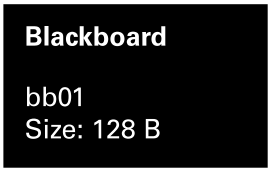
Message Pattern

Working with resources like ports, blackboards, and buffers requires messages which can be written to and read from them. For testing purposes the actual contents are not relevant, therefore it is sufficient to have message patterns.

Properties	Message payload, either as concrete values or as sets of allowed values, e. g. regular expressions.
Relationships	Connected to message-related API calls or complement operations.
Rules	None.
Representation	

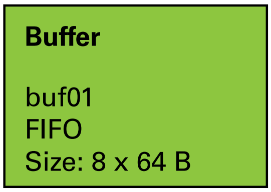
Blackboard

ARINC 653 blackboards can be created at runtime and can be used for intra-partition communication.

Properties	The name of the blackboard. The maximum size of a message.
Relationships	Connected to blackboard-related API calls.
Rules	None.
Representation	

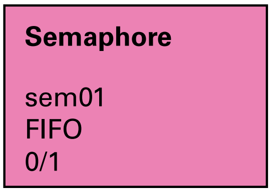
Buffer

ARINC 653 buffers can be created at runtime and can be used for intra-partition communication.

Properties	The name of the buffer. The queuing discipline (FIFO or priority). The maximum size of a message. The maximum number of messages.
Relationships	Connected to buffer-related API calls.
Rules	None.
Representation	

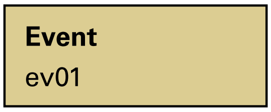
Semaphore

ARINC 653 semaphores can be created at runtime. They provide an intra-partition synchronisation mechanism (counting semaphore) with a maximum counter value.

Properties	The name of the semaphore. The queuing discipline (FIFO or priority). The initial counter value. The maximum counter value.
Relationships	Connected to semaphore-related API calls.
Rules	None.
Representation	

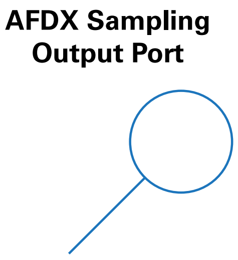
Event

ARINC 653 events can be created at runtime. They provide a synchronisation mechanism that allows a process to signal the occurrence of an event to one or more other waiting processes.

Properties	The name of the event object.
Relationships	Connected to event-related API calls.
Rules	None.
Representation	

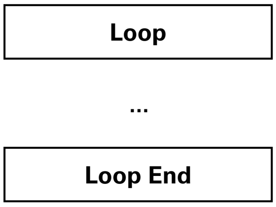
Port

A port represents an API port defined in the partition configuration. It is of a specific type and direction. In a loop (see below) it will represent each configured port matching the characteristics in turn.

Properties	The type of port (AFDX or RAM). The port characteristic (sampling or queuing). The direction of the port (input or output).
Relationships	Connected to port-related API calls. Connected to a loop object.
Rules	Must be connected to exactly one loop object.
Representation	

Loop

In order to have tests that are not tailored to a specific module configuration, it is necessary to have a means of working with an arbitrary number of ports defined in a configuration. This is possible with a loop construct, performing actions inside the loop for each port defined in the configuration. A loop consists of two blocks, a loop start and a loop end, bracketing the loop body objects.

Properties	None.
Relationships	Connections to and from API calls, processes, or complements. Connected to a port object.
Rules	Must be connected to exactly one port object.
Representation	

9.1.3 Bare Module Testing: Test Suite

Finally, a mechanism to associate configurations and test behaviours is required, i. e. a *test suite*, in which all relevant configurations and behaviours of a particular test project are grouped together.

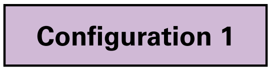
Behaviour

A Behaviour represents an entire test behaviour model as described in section 9.1.2. Each test behaviour has one or more links to configurations with which the test shall be executed.

Properties	The referenced Test Behaviour model.
Relationships	Connected to one or more Configurations.
Rules	None.
Representation	AFDX Port Test 1

Configuration

A Configuration represents an entire configuration model as described in section 9.1.1. Each configuration can have links to one or more behaviours with which the configuration is to be employed.

Properties	The referenced Test Configuration model.
Relationships	Connected to one or more Behaviours.
Rules	None.
Representation	

9.1.4 Configured Module Testing

As described before in section 2.4, Configured Module Testing involves testing the IMA module, its operating system, and the designated configuration for the module, generated by the module integrator for the module's intended operation mode (i. e. the specific aircraft configuration).

When analysing the Configured Module Testing usage domain, it becomes apparent that it bears no new modelling requirements compared to Bare Module Testing. Instead, module configurations are already provided and part of the SUT now, so all that is required is to be able to use behaviour specifications as described in section 9.1.2 with pre-existing module configurations.

This is fulfilled by the test generator (cf. section 10.2.2) processing the actual module configuration files (ICDs in CSV format), rather than only working on module configuration models as described in section 9.1.1.

9.1.5 Hardware/Software Integration Testing

The purpose of Hardware/Software Integration Testing is to verify the correct behaviour of software implementing an actual avionics function (like flight control, fire and smoke detection, or braking control systems) on the IMA module hardware.

We need to be able to model the ideal (i. e. correct) behaviour of the application software, so that the actual behaviour of the implementation can be checked against the model. Since, at this level of testing, the module and its software is considered a black box (i. e. we cannot examine its internal state during testing), the only observable evidence of its behaviour are the outputs it produces via its I/O interfaces as reaction to input stimuli.

Based on this, we wish to specify the correct behaviour by modelling the internal states of the system and how it calculates its outputs based on the inputs from its environment. At the top level, we have the SUT and its environment, connected via the SUT's input and output signals. Furthermore, the SUT may consist of an arbitrary number of internal functional blocks, called components. On the lowest level, we wish to model the behaviour of a component in a formal way, e. g. by specifying statecharts. As described earlier in chapter 4, it is highly desirable to have extended state variables in order to avoid state explosion. Such variables can have different data types. Since we wish to model real-time behaviour, we obviously need clock variables as well.

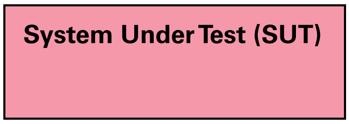
The individual locations inside a statechart are connected via transitions. A transition can have a guard condition and an action, while a location can have entry, do, and exit actions. Each of these types of action can change internal variables and/or externally visible output signals, while guard conditions control state transitions depending on internal variables and/or external input signals.

In contrast to the other language variants, here we would like to express a hierarchy (of components). As briefly mentioned in section 3.2.2, this is possible by employing a so-called *decomposition*. This means that an object is linked to a (sub-)graph, and this graph shows the interior of the object.

The aforementioned elements are sufficient for the modelling of system behaviour. We can now give a list summarising the concepts of the application domain with their designated properties, relationships, constraint rules, and their representation.

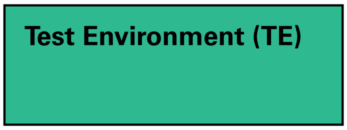
SUT

The SUT represents the module and its software as a whole. It is the top-level component for modelling the module's behaviour.

Properties	The name of the SUT.
Decomposition	To Component Diagram or Statechart.
Rules	Must occur exactly once.
Representation	

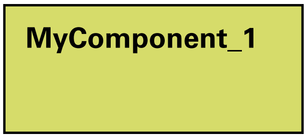
TE

The TE represents the Test Environment, i. e. the counterpart to the SUT. Like the SUT, it is a top-level component, but the TE is used to model how the environment behaves.

Properties	The name of the TE.
Decomposition	To Component Diagram or Statechart.
Rules	Must occur exactly once.
Representation	

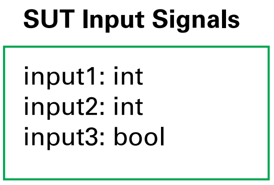
Component

A component represents a functional block in either the SUT or the TE.

Properties	The name of the component.
Decomposition	To Component Diagram or Statechart.
Rules	None.
Representation	

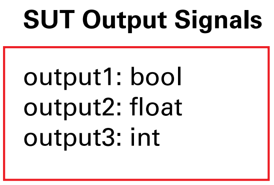
Input Signals List

The input signals list specifies all signals which the TE can send as inputs to the SUT.

Properties	List of signal definitions.
Relationships	None.
Rules	The names of all input and output signals must be unique.
Representation	 SUT Input Signals input1: int input2: int input3: bool

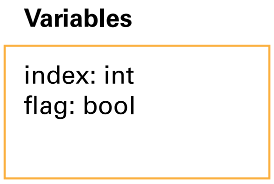
Output Signals List

The output signals list specifies all signals which the SUT can send as outputs to the TE.

Properties	List of signal definitions.
Relationships	None.
Rules	The names of all input and output signals must be unique.
Representation	 SUT Output Signals output1: bool output2: float output3: int

Variables List

A variables list defines variables inside a component, which make up the component's extended state.

Properties	List of variable definitions.
Relationships	None.
Rules	The names of all variables and constants within a component must be unique.
Representation	

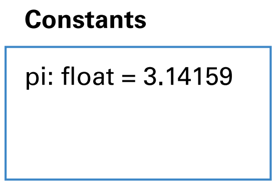
Variable Definition

A variable definition specifies a single variable.

Properties	Name of the variable. Type of the variable. Default value. Minimum value. Maximum value.
Relationships	None.
Rules	None.
Representation	n/a

Constants List

A constants list defines constants, i. e. symbolic names for constant numeric values.

Properties	List of constant definitions.
Relationships	None.
Rules	The names of all variables and constants within a component must be unique.
Representation	

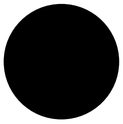
Constant Definition

A constant definition specifies a single constant.

Properties	Name of the constant. Type of the constant. Value of the constant.
Relationships	None.
Rules	None.
Representation	n/a

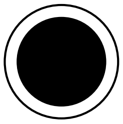
Start Location

The start location designates the initial location of a statechart.

Properties	None.
Relationships	Connected to location.
Rules	Must occur exactly once per statechart.
Representation	

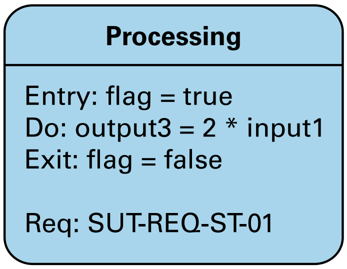
Stop Location

The stop location designates a terminal location within a statechart.

Properties	None.
Relationships	Connected to locations.
Rules	None.
Representation	

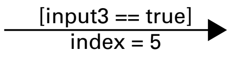
Location

A location describes a state within a statechart. It has lists of actions, which can, for example, be assignments or other code statements.

Properties	The name of the location. List of entry actions. List of do actions. List of exit actions. Requirement tag.
Relationships	Connected to locations.
Rules	None.
Representation	 Processing Entry: flag = true Do: output3 = 2 * input1 Exit: flag = false Req: SUT-REQ-ST-01

Transition

A transition is the link between locations in a statechart. It has an outgoing location and an ingoing location. In the graphical representation, the arrow points at the ingoing location.

Properties	The name of the transition. A condition for the transition. List of actions. Requirement tag.
Relationships	Outgoing location can be Start Location or Location. Ingoing location can be Location or Stop Location.
Rules	A transition has exactly one outgoing and one ingoing location.
Representation	

9.1.6 Design Considerations

After completing the domain analysis, it is now possible to specify the new language. As we can see, the language shall cover three different usage scenarios (bare module, configured module, and hardware/software integration). Adhering to the fourth rule from section 9.1, instead of covering all at once, the language is split into three variants:

ITML-B Bare Module testing. This variant allows the definition of sets of configurations as well as the specification of test case behaviour.

ITML-C Configured Module testing. This variant does not permit the definition of configurations, but supports the specification of test case behaviour.

ITML-A Hardware/Software Integration testing. This variant allows the specification of application behaviour, from which test cases can be generated automatically.

The following sections describe the implementation of the ITML language within the domain-specific modelling framework MetaEdit+ (cf. section 3.2). Each variant has an abstract syntax definition, as well as static and dynamic semantics. The abstract syntax is provided in the form of a graphical representation of GOPPRR metamodels. They show the types of objects that make up a graph, their respective properties, and the relationships and respective roles connecting the objects. While constraints are not visible in these diagrams, they are provided in textual form, defining the static semantics of the language.

MetaEdit+ has direct support for several types of constraints, i. e. its framework guarantees that the models created with it always comply to the constraints defined in the metamodel. It is not possible to perform a modelling action that would result in an invalid model. As analysed in [Mew10, ch. 4.4], this includes connectivity constraints (maximum number of roles or relationships an object can be in), occurrence constraints (maximum number of occurrences in a graph for an object), port constraints (same or different values of properties of all objects in a binding), and uniqueness constraints (uniqueness of property values per graph).

It is, however, important to note that the constraints supported by MetaEdit+ can only limit the *maximum* number of occurrences or connections, not the *minimum*. It is for example possible to say "Objects of type x can be in at most 2 y relationships", but it is not possible to say "Objects of type x must be in at least 1 y relationship". The reason is that, were it possible to define such a constraint, the model could not always be compliant: It is not possible to create a new object and put it into all the required relationships in an atomic operation.

Nonetheless, the generator facilities provided by MetaEdit+ can be employed to write a model validity checker. This checker can traverse the model and check the model's adherence to all additional constraints. It can, for example, count the number of relationships that each object of type x is in, and display an error message to the user if this number is below the required minimum.

9.2 ITML-B: Bare Module Testing

The IMA Test Modelling Language variant B enables the test expert to define bare module tests. Since there are no pre-defined configurations for bare module testing, the language allows the definition of suitable configurations, in addition to API-oriented test actions. Therefore, ITML-B is split into three parts: The configuration modelling part allows the definition of a set of configurations for a test case, while the behaviour modelling part allows the specification of the dynamic test case behaviours. The test suite part groups them together and makes the connection between behaviours and configurations.

Each of these parts is realised as a graph type in MetaEdit+.

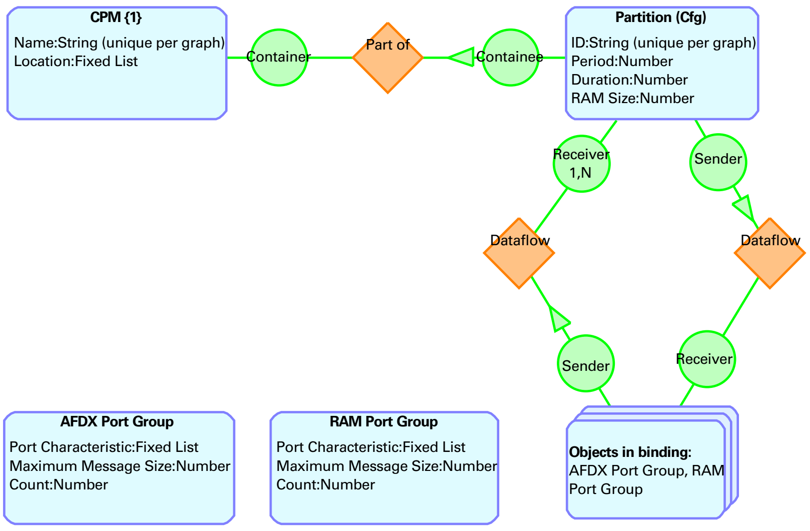


Figure 9.1: Abstract Syntax: Configuration Part

9.2.1 Configuration

The goal of the Configuration part is not to provide a full-featured configuration editor, but to enable the user to describe constraints on a set of configurations with a minimum of effort. The Configuration part is represented by a graph that comprises the different components which make up a CPM configuration.

Abstract Syntax

Figure 9.1 shows the abstract syntax of the configuration part. Object types are represented by blue rounded rectangles. Their properties are displayed inside the rectangle body. Objects are connected via relationships, displayed as orange diamonds. A green circle shows the role which an object has in a relationship.

The root object is a CPM. The module-level configuration parameters relevant for Bare Module testing are represented as properties of this object[2].

One or more partition objects can be associated with a CPM object. The partition objects are in a *Part of* relationship with the CPM. The partition-level configuration parameters relevant for Bare Module testing are represented as properties of this object. Each configured partition will execute the Test Agent. Appropriate command ports will be defined automatically.

[2]Other (static) configuration parameters are provided from a template.

AFDX Port Group objects and RAM Port Group objects can be in *Dataflow* relationships with a partition.

RAM port group objects have associations with two partitions. The associations have an arrow to indicate the port direction. The object has properties that represent the range of RAM port configuration parameters. That means a RAM port group object does not represent one single RAM port, but a set of different RAM ports, generated automatically by the configuration generator.

AFDX port group objects have an association with one partition. The association has an arrow to indicate the port direction. The object has properties that represent the range of AFDX port configuration parameters. That means an AFDX port group object does not represent one single AFDX port, but a set of different AFDX ports, generated automatically by the configuration generator.

The concrete syntax of the language elements defines their actual appearance in models. Appropriate symbols have been defined in MetaEdit+ to have the elements look like their representations as shown during the domain analysis (cf. section 9.1).

Relationships and roles have a concrete syntax as well. The *Part of* relationship is represented by a solid line. The *Receiver* role of the *Dataflow* relationship has an incoming arrowhead representation, thus denoting the direction of dataflow in the relationship.

Static Semantics

As determined during the domain analysis, a Configuration graph must adhere to the following constraints:

1. Property “Name” in CPM must have unique values.
2. Property “Location” in CPM must have unique values.
3. Partition may be in at most one Containee role.
4. Property “ID” in Partition must have unique values.
5. AFDX Port Group may be in at most one Dataflow relationship.
6. RAM Port Group may be in at most two Dataflow relationships.
7. RAM Port Group may be in at most one Sender role.
8. RAM Port Group may be in at most one Receiver role.

While MetaEdit+ directly allows the incorporation of the preceding constraints into the ITML metamodel (thus guaranteeing model compliance), the remaining constraints cannot be checked automatically by MetaEdit+ and must instead be checked by a model validity checker:

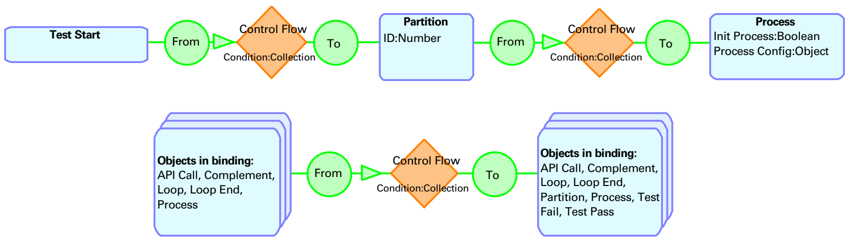


Figure 9.2: Abstract Syntax: Behaviour Control Flow

9. The partition slice duration must be less than the partition scheduling period.
10. An AFDX Port Group must be in exactly one Dataflow relationship.
11. A RAM Port Group must be in exactly one Sender role.
12. A RAM Port Group must be in exactly one Receiver role.
13. A Partition must be in exactly one Containee role.

The corresponding model validity checker can be found in appendix D.1.

9.2.2 Behaviour

The Behaviour part is a different graph type. It comprises flowchart-like components.

Abstract Syntax

Due to its size the abstract syntax is split across several figures. Figure 9.2 shows the control-flow-related parts of the abstract syntax. The execution flow is represented by objects connected by *Control Flow* relationships (arrows). Each relationship can have an optional guard condition.

A behaviour part graph has exactly one flow start object and one or more flow end objects, each denoting the test case result (pass/fail).

To give the context, i. e. partition and process, where API calls are to be executed, a sequence of API calls must be preceded by a partition object and a process object. A partition object defines the partition, a process object defines the process inside a partition, by which the following sequence of API calls is to be executed.

Figure 9.3 presents the different types of nodes in the control flow graph. Most of them are API calls. The concrete API call objects are derived from the abstract API call object type.

The steps in the behaviour part are API call objects which are to be executed by the Test Agent in one or more processes. The API call parameters

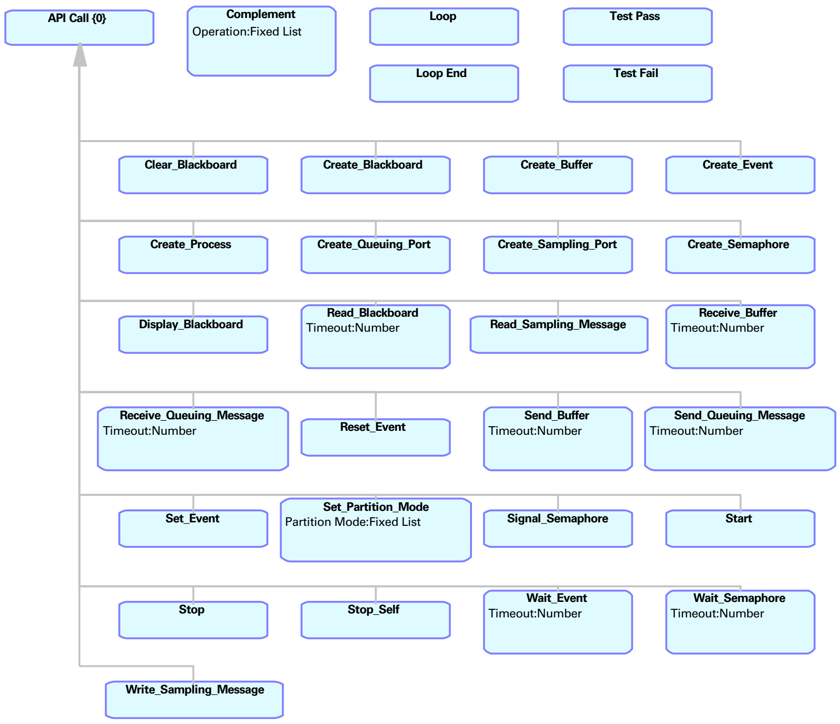


Figure 9.3: Behaviour Metamodel: Nodes

are derived from parameter objects, like an API port or a blackboard. The *Parameter* relationships allowed between API calls and resource objects can be seen in figure 9.4: The different kinds of API calls require different kinds of parameter objects. A *Create_Event* API call, for example, must be connected to an *Event* object, but it cannot be connected to a *Buffer* object. Some API calls require two or more parameter objects of different types. For each kind of parameter object there is a different role which the API call object assumes in that relationship. For example, a *Send_Buffer* API call will be in an *API call [Buffer]* role in its relationship to the *Buffer* parameter object, while at the same time it will be in an *API call [Message]* role in its relationship to the *Message* parameter object. These different roles are necessary to create constraints that ensure API calls have the required number of different parameters (cf. Static Semantics below).

Read or write *Complement* steps allow access to the other end of a port resource currently in use without having to know (and hard-code) where (e. g. in which partition, or on which external AFDX port) that is actually located.

Message pattern objects specify message payloads, either as concrete values or as sets of allowed values, e. g. regular expressions.

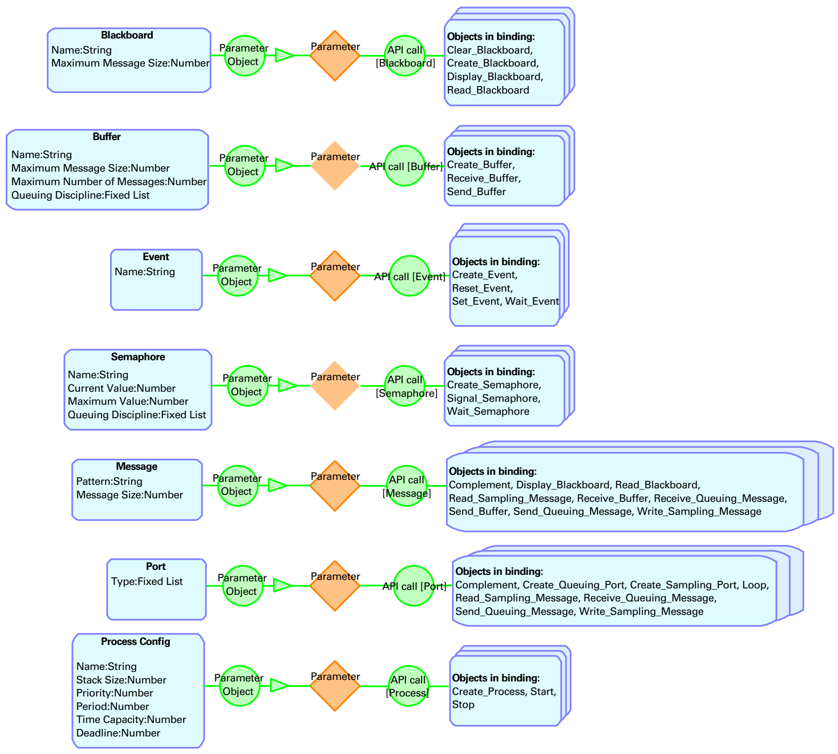

Blackboard
Name:String
Maximum Message Size:Number
Parameter Object
Parameter
API call [Blackboard]
Objects in binding: Clear_Blackboard, Create_Blackboard, Display_Blackboard, Read_Blackboard
Buffer
Name:String
Maximum Message Size:Number
Maximum Number of Messages:Number
Queuing Discipline:Fixed List
API call [Buffer]
Objects in binding: Create_Buffer, Receive_Buffer, Send_Buffer
Event
Name:String
API call [Event]
Objects in binding: Create_Event, Reset_Event, Set_Event, Wait_Event
Semaphore
Name:String
Current Value:Number
Maximum Value:Number
Queuing Discipline:Fixed List
API call [Semaphore]
Objects in binding: Create_Semaphore, Signal_Semaphore, Wait_Semaphore
Message
Pattern:String
Message Size:Number
API call [Message]
Objects in binding: Complement, Display_Blackboard, Read_Blackboard, Read_Sampling_Message, Receive_Buffer, Receive_Queuing_Message, Send_Buffer, Send_Queuing_Message, Write_Sampling_Message
Port
Type:Fixed List
API call [Port]
Objects in binding: Complement, Create_Queuing_Port, Create_Sampling_Port, Loop, Read_Sampling_Message, Receive_Queuing_Message, Send_Queuing_Message, Write_Sampling_Message
Process Config
Name:String
Stack Size:Number
Priority:Number
Period:Number
Time Capacity:Number
Deadline:Number
API call [Process]
Objects in binding: Create_Process, Start, Stop

Figure 9.4: Behaviour Metamodel: Parameters

Resource objects are used to hold references to OS resources created at runtime (e. g. blackboards). They can be associated with the applicable API call objects as parameter objects.

Behaviour steps can be enclosed in loop start and loop end objects. This shall allow the specification of steps that are to be executed for all port resources of a specified type.

Static Semantics

Again, as determined during the domain analysis, a Behaviour graph must adhere to the following constraints:

1. Test Start may occur at most one time.
2. Clear_Blackboard may be in at most one API call [Blackboard] role.
3. Create_Blackboard may be in at most one API call [Blackboard] role.
4. Create_Buffer may be in at most one API call [Buffer] role.

5. Create_Event may be in at most one API call [Event] role.
6. Create_Process may be in at most one API call [Process] role.
7. Create_Queuing_Port may be in at most one API call [Port] role.
8. Create_Sampling_Port may be in at most one API call [Port] role.
9. Create_Semaphore may be in at most one API call [Semaphore] role.
10. Display_Blackboard may be in at most one API call [Blackboard] role.
11. Display_Blackboard may be in at most one API call [Message] role.
12. Read_Blackboard may be in at most one API call [Blackboard] role.
13. Read_Blackboard may be in at most one API call [Message] role.
14. Read_Sampling_Message may be in at most one API call [Message] role.
15. Read_Sampling_Message may be in at most one API call [Port] role.
16. Receive_Buffer may be in at most one API call [Buffer] role.
17. Receive_Buffer may be in at most one API call [Message] role.
18. Receive_Queuing_Message may be in at most one API call [Message] role.
19. Receive_Queuing_Message may be in at most one API call [Port] role.
20. Reset_Event may be in at most one API call [Event] role.
21. Send_Buffer may be in at most one API call [Buffer] role.
22. Send_Buffer may be in at most one API call [Message] role.
23. Send_Queuing_Message may be in at most one API call [Message] role.
24. Send_Queuing_Message may be in at most one API call [Port] role.
25. Set_Event may be in at most one API call [Event] role.
26. Signal_Semaphore may be in at most one API call [Semaphore] role.
27. Start may be in at most one API call [Process] role.
28. Stop may be in at most one API call [Process] role.
29. Wait_Event may be in at most one API call [Event] role.
30. Wait_Semaphore may be in at most one API call [Semaphore] role.

31. Write_Sampling_Message may be in at most one API call [Message] role.
32. Write_Sampling_Message may be in at most one API call [Port] role.
33. Complement may be in at most one API call [Message] role.
34. Complement may be in at most one API call [Port] role.

While MetaEdit+ directly supports the previous constraints, the remaining constraints cannot be checked automatically by MetaEdit+ and must instead be checked by a model validity checker:

35. Test Start must occur exactly one time.
36. Clear_Blackboard must be in exactly one API call [Blackboard] role.
37. Create_Blackboard must be in exactly one API call [Blackboard] role.
38. Create_Buffer must be in exactly one API call [Buffer] role.
39. Create_Event must be in exactly one API call [Event] role.
40. Create_Process must be in exactly one API call [Process] role.
41. Create_Queuing_Port must be in exactly one API call [Port] role.
42. Create_Sampling_Port must be in exactly one API call [Port] role.
43. Create_Semaphore must be in exactly one API call [Semaphore] role.
44. Display_Blackboard must be in exactly one API call [Blackboard] role.
45. Display_Blackboard must be in exactly one API call [Message] role.
46. Read_Blackboard must be in exactly one API call [Blackboard] role.
47. Read_Blackboard must be in exactly one API call [Message] role.
48. Read_Sampling_Message must be in exactly one API call [Message] role.
49. Read_Sampling_Message must be in exactly one API call [Port] role.
50. Receive_Buffer must be in exactly one API call [Buffer] role.
51. Receive_Buffer must be in exactly one API call [Message] role.
52. Receive_Queuing_Message must be in exactly one API call [Message] role.
53. Receive_Queuing_Message must be in exactly one API call [Port] role.

54. Reset_Event must be in exactly one API call [Event] role.
55. Send_Buffer must be in exactly one API call [Buffer] role.
56. Send_Buffer must be in exactly one API call [Message] role.
57. Send_Queuing_Message must be in exactly one API call [Message] role.
58. Send_Queuing_Message must be in exactly one API call [Port] role.
59. Set_Event must be in exactly one API call [Event] role.
60. Signal_Semaphore must be in exactly one API call [Semaphore] role.
61. Start must be in exactly one API call [Process] role.
62. Stop must be in exactly one API call [Process] role.
63. Wait_Event must be in exactly one API call [Event] role.
64. Wait_Semaphore must be in exactly one API call [Semaphore] role.
65. Write_Sampling_Message must be in exactly one API call [Message] role.
66. Write_Sampling_Message must be in exactly one API call [Port] role.
67. Complement must be in exactly one API call [Message] role.
68. Complement must be in exactly one API call [Port] role.
69. Conditions on outgoing Control Flow edges of a node must be deterministic.
70. Nodes outside a Loop block must not have Control Flow relationships to nodes inside a Loop block.
71. Nodes having a Control Flow relationship to a Loop End block must be inside a loop.

The corresponding model validity checker can be found in appendix D.2.

9.2.3 Test Suite

The third graph type is the Test Suite. It combines several Configuration and Behaviour parts.

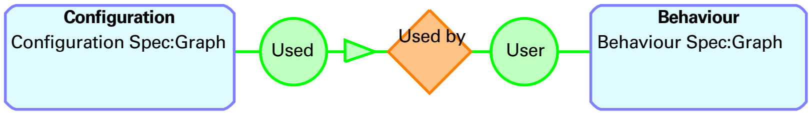


Figure 9.5: ITML-B Test Suite

Abstract Syntax

The abstract syntax of the Test Suite graph type is depicted in figure 9.5. Here, a Behaviour object is always associated with one or more configuration objects. Therefore, the graph specifies for each test case that is part of the test suite the configurations with which it is to be executed.

Both Configuration and Behaviour objects have a property which directly links to the concrete graph instance which the object represents.

Static Semantics

The relative simplicity of the Test Suite graph also reflects on its constraints. Each graph must be compliant with the following two constraints:

1. A Behaviour must be in at least 1 Used by relationship.

2. A Configuration must be in at least 1 Used by relationship.

Since these constraints cannot be enforced directly by MetaEdit+, they must be checked by a model validity checker. The corresponding model validity checker can be found in appendix D.3.

9.2.4 Dynamic Semantics

For each graph type introduced until now, the previous sections gave definitions of syntax and static semantics, but a definition of dynamic semantics was still missing. The reason for this is that the behaviour can only be defined when taking all graphs into consideration. In particular, the semantics of a Behaviour graph can only be determined in combination with a Configuration. This combination is made explicit in the Test Suite graph.

In order to precisely define the semantics of MetaEdit+ graphs, some auxiliary constructs are required:

Gra_{type}	set of all graphs of type *type*
$Obj_{type}(G)$	set of objects of type *type* in graph G
$Prop_{name}(o)$	value of property *name* of object o
$Prop_{name}(r)$	value of property *name* of relationship r
$Rel_{type}(G, o_1, o_2)$	set of relationships of type *type* between objects o_1 and o_2 in graph G
$RelO_{type}(G, o)$	set of objects in relationship of type *type* to object o in graph G
$Role_{type}(G, o)$	set of objects in role of type *type* in a relationship to object o in graph G

From each Test Suite graph $t \in Gra_{TestSuite}$ a set TP of test procedures can be derived. The set of test procedures TP contains pairs of configuration and behaviour graphs:

$$\begin{aligned}\langle c, b\rangle \in TP \Leftrightarrow\ &\exists c' \in Obj_{Configuration}(t)\ \ \exists b' \in Obj_{Behaviour}(t)\ \mid \\ &c = Prop_{ConfigSpec}(c')\ \wedge\ b = Prop_{BehaviourSpec}(b')\ \wedge\ c' \in RelO_{Usedby}(t, b')\end{aligned}$$

A test procedure tuple $\langle c, b\rangle$ is element of the test procedure set TP iff its Test Suite graph t contains objects c' and b' referencing graphs c and b, and c' is in a relationship with b'.

What remains now is the semantics of a single test procedure, made up of Configuration graph c and Behaviour graph b. The observable effect of a test procedure must be a sequence of ARINC 653 API calls:

$$a_1 \rightarrow a_2 \rightarrow a_3 \rightarrow \dots \rightarrow a_n$$

Each call a_i has input and output parameters, a return value, and is addressed to (a Test Agent in) a specific partition and process. For easy and intuitive representation, we define a call to have the following format:

$$Partition : Process : API\ Call\ (\ Parameters,\ Return\ Value\ )$$

The call `9:5:SET_PARTITION_MODE(WARM_START, NO_ERROR)`, for example, means: "Process 5 in partition 9 makes a Set_Partition_Mode API call with parameter Warm_Start. It returns with code No_Error."

We will use a structural operational semantics approach [Plo81] to describe how this sequence of calls is produced from a test procedure. To that end, we define a variant of a labelled transition system, called a *behaviour transition system*, as well as transition rules on the state of the transition system.

Definitions

Let V be the set of possible test verdicts and R be the set of possible API call return values:

$$\begin{aligned} V &= \{\text{pass}, \text{fail}, \text{unknown}\} \\ R &= \{\text{NO\_ERROR}, \text{NO\_ACTION}, \text{NOT\_AVAILABLE}, \text{INVALID\_PARAM}, \\ &\quad\ \text{INVALID\_CONFIG}, \text{INVALID\_MODE}, \text{TIMED\_OUT}\} \end{aligned}$$

Let P_a be the set of all partition numbers and P_r be the set of all process indices:

$$\begin{aligned} P_a &= \{1 \dots 32\} \\ P_r &= \{0 \dots 2^{32} - 1\} \end{aligned}$$

Let O be the set of parameter objects, L be the set of locations in a Behaviour graph b, and Φ be the set of all loop start locations plus a "no-loop" element $\perp$:

$$\begin{aligned} O &= Obj_{Blackboard}(b) \cup Obj_{ProcessConfig}(b) \cup \dots \\ L &= Obj_{TestStart}(b) \cup Obj_{Partition}(b) \cup Obj_{Process}(b) \cup \dots \\ \Phi &= Obj_{Loop}(b) \cup \{\perp\} \end{aligned}$$

Let Π be the set of all API ports in a configuration c, and let $Port(p_a, t, n)$ return the n-th API port object of type t in the configuration of partition p_a. (**Note**: Here we are not talking about *port groups* as defined earlier, but about individual API ports. While this definition may seem slightly vague, it is highly convenient as we can later reuse it for ITML-C. Furthermore, we handle API ports in the same way as graph objects, so that we can access their properties with *Prop*.)

Let M be a set of memory functions $m : O \cup \Pi \to \mathbb{N}$ (cf. [Plo81, p. 15]). A memory m stores an ID value for each parameter object and port. In order to update stored values, we define for a given memory m, a parameter object o, and a natural number i the memory $m' = m[i/o]$ where

$$m'(o') = \begin{cases} i & \text{if } o' = o, \\ m(o') & \text{otherwise.} \end{cases}$$

Now we define a behaviour transition system B as a 5-tuple:

$$B = \langle S, \Lambda, T, P, s_0 \rangle$$

where

$S = L \times \Phi \times M \times R \times V \times P_a \times P_r$	state space,
$\Lambda = \mathscr{P}(R)$	labels,
$T \subseteq L \times \Lambda \times L$	transition relation,
$P \subseteq L \times O$	parameter relation,
$s_0 \in S$	initial state.

The transition relation T contains tuples representing the control flow connections in the Behaviour graph b:

$$\begin{aligned}\langle l, \lambda, l' \rangle \in T \Leftrightarrow l' \in RelO_{ControlFlow}(b, l) \cap Role_{To}(b, l) \wedge \\ \lambda = Prop_{Condition}(Rel_{ControlFlow}(b, l, l'))\end{aligned}$$

A transition tuple $\langle l, \lambda, l' \rangle$ is element of the transition relation iff the Behaviour graph contains a Control Flow relationship from l to l' and that relationship has the set of return values, λ, in its condition property. Furthermore, in order to simplify the state transition rules below, we define that for a transition tuple $\langle l, \lambda, l' \rangle$ whose relationship has an empty condition (default transition), its label λ shall be set to the set of all return values R without those return values occurring in other relationships outgoing from l:

$$\lambda = R \setminus \left(\bigcup Prop_{Condition}(Rel_{ControlFlow}(b, l, l')) \;\mid\; l' \in Role_{To}(b, l) \right)$$

This definition is legal due to rule no. 69 of the static semantics, which prohibits more than one default transition as well as multiple occurrences of a return value in outgoing transitions from a single location.

The parameter relation P contains tuples representing parameter associations between control locations (API calls etc.) and parameter objects:

$$\langle l, o \rangle \in P \Leftrightarrow \; o \in RelO_{Parameter}(b, l)$$

A parameter tuple $\langle l, o \rangle$ is element of the parameter relation iff the Behaviour graph contains a Parameter relationship between location l and object o.

Initial State

Next, we need to give a definition of the initial state s_0 of the behaviour transition system. For a given Behaviour graph b, the initial state is the 7-tuple

$$s_0 = \langle l_0, \varphi_0, m_0, r_0, v_0, p_{a0}, p_{r0} \rangle$$

where

$l_0 \in Obj_{TestStart}(b)$	start location,
$\varphi_0 = \perp$	no current loop,
$m_0(o) = 0 \ \ \forall\, o \in O$	zero-initialised memory,
$r_0 = \text{NO\_ERROR}$	initial return code,
$v_0 = \text{unknown}$	initial test verdict,
$p_{a0} = 1$	initially addressed partition,
$p_{r0} = 0$	initially addressed process.

In particular, the start location l_0 is set to the only[3] Test Start object in the graph. The definition of m_0 assigns the value 0 to all parameter objects. Furthermore, the assignments of p_{a0} and p_{r0} just provide initial dummy values, as the abstract syntax requires at least one partition and process object before API calls in the graph (cf. figure 9.2).

Behaviour Transition Rules

Finally, we need a set of transition rules, which define how the state of the behaviour transition system changes, thereby producing a sequence of API calls. We write $s \xrightarrow{a} s'$ to denote that state s transitions to state s', producing API call a. If a transition does not result in an API call, we simply write $s \longrightarrow s'$ instead.

If a transition rule is possible, it must be applied. If more than one transition rule can be applied in the current state, the rule with the lowest number must be applied. If no rule can be applied in the current state, the test procedure terminates with v as the final test verdict.

Transition rule 1:

$$\frac{\langle l, \lambda, l' \rangle \in T \wedge r \in \lambda \wedge l' \in Obj_{Partition}(b)}{\langle l, \varphi, m, r, v, p_a, p_r \rangle \longrightarrow \langle l', \varphi, m, r, v, Prop_{ID}(l'), p_r \rangle}$$

The first rule defines the change of state if from the current location l a control flow connection to a Partition object l' exists and the condition λ on that connection permits this transition. The successor state then has l' as the current location and the value of the Partition object's ID property as the currently addressed partition.

Transition rule 2:

$$\frac{\langle l, \lambda, l' \rangle \in T \wedge r \in \lambda \wedge l' \in Obj_{Process}(b) \wedge Prop_{InitProcess}(l')}{\langle l, \varphi, m, r, v, p_a, p_r \rangle \longrightarrow \langle l', \varphi, m, r, v, p_a, 0 \rangle}$$

[3]as per static semantics rule no. 35

This rule defines the change of state if from the current location l a control flow connection to a Process object l' indicating the init process exists and the condition λ on that connection permits this transition. The successor state then has l' as the current location and the value 0[4] as the currently addressed process.

Transition rule 3:

$$\frac{\langle l,\lambda,l'\rangle \in T \wedge r \in \lambda \wedge l' \in Obj_{Process}(b) \wedge \neg Prop_{InitProcess}(l')}{\langle l,\varphi,m,r,v,p_a,p_r\rangle \longrightarrow \langle l',\varphi,m,r,v,p_a,m(Prop_{ProcessConfig}(l'))\rangle}$$

This rule defines the change of state if from the current location l a control flow connection to a Process object l' *not* referencing the init process exists and the condition λ on that connection permits this transition. The successor state then has l' as the current location and the memory-stored index value of the process configuration object[5] referenced by the Process object l' as the currently addressed process.

Transition rule 4:

$$\frac{\langle l,\lambda,l'\rangle \in T \wedge r \in \lambda \wedge l' \in Obj_{TestFail}(b)}{\langle l,\varphi,m,r,v,p_a,p_r\rangle \longrightarrow \langle l',\varphi,m,r,\mathrm{fail},p_a,p_r\rangle}$$

This rule defines the change of state if from the current location l a control flow connection to a Test Fail object l' exists and the condition λ on that connection permits this transition. The successor state then has l' as the current location and the value “fail” as the new test verdict.

Transition rule 5:

$$\frac{\langle l,\lambda,l'\rangle \in T \wedge r \in \lambda \wedge l' \in Obj_{TestPass}(b) \wedge v \neq \mathrm{fail}}{\langle l,\varphi,m,r,v,p_a,p_r\rangle \longrightarrow \langle l',\varphi,m,r,\mathrm{pass},p_a,p_r\rangle}$$

This rule defines the change of state if from the current location l a control flow connection to a Test Pass object l' exists and the condition λ on that connection permits this transition. The successor state then has l' as the current location and the value “pass” as the new test verdict. However, this transition is only allowed, as can be seen in the premise, if the previous verdict is not “fail”, as a test that has already failed cannot be passed any more.

Transition rule 6:

$$\frac{\langle l,\lambda,l'\rangle \in T \wedge r \in \lambda \wedge l' \in Obj_{Create\_Blackboard}(b) \wedge \langle l',o\rangle \in P}{\langle l,\varphi,m,r,v,p_a,p_r\rangle \xrightarrow[Prop_{MaxMsgSize}(o),i,r')]{p_a:p_r:\texttt{Create\_Blackboard}(Prop_{Name}(o),} \langle l',\varphi,m[i/o],r',v,p_a,p_r\rangle}$$

[4]the index of the init process as expected by the test agent

[5]This happens during the *Create_Process* API call, similar to the Blackboard ID in transition rule 6.

This rule defines the change of state if from the current location l a control flow connection to a Create_Blackboard object l' associated with a Blackboard parameter object o exists and the condition λ on the control flow connection permits this transition. This transition results in a Create_Blackboard API call addressed to the current process p_r in the current partition p_a, with its input parameters taken from the appropriate properties of parameter object o, its output parameter i, and its return value r'. The successor state then has l' as the current location, r' as the last return value, and memory m is updated to contain the returned blackboard ID i as value for Blackboard object o.

Transition rule 7:

$$\frac{\langle l,\lambda,l'\rangle \in T \land r \in \lambda \land l' \in Obj_{Clear\_Blackboard}(b) \land \langle l',o\rangle \in P}{\langle l,\varphi,m,r,v,p_a,p_r\rangle \xrightarrow{p_a:p_r:\texttt{Clear\_Blackboard}(m(o),r')} \langle l',\varphi,m,r',v,p_a,p_r\rangle}$$

This rule defines the change of state if from the current location l a control flow connection to a Clear_Blackboard object l' associated with a Blackboard parameter object o exists and the condition λ on the control flow connection permits this transition. This transition results in a Clear_Blackboard API call addressed to the current process p_r in the current partition p_a, with the blackboard ID for parameter object o stored in memory m as its input parameter, and its return value r'. The successor state then has l' as the current location and r' as the last return value.

Additional transition rules must be applied for each of the other API call objects. However, due to their similarity to the two last rules, they are very easy to deduce and thus will not be reproduced here for the sake of brevity. Instead, we will take a look at the transition rules for loop handling.

Transition rule 8:

$$\frac{\langle l,\lambda,l'\rangle \in T \land r \in \lambda \land l' \in Obj_{Loop}(b) \land \langle l',o\rangle \in P}{\langle l,\varphi,m,r,v,p_a,p_r\rangle \longrightarrow \langle l',l',m[0/o],r,v,p_a,p_r\rangle}$$

This rule defines the change of state if from the current location l a control flow connection to a Loop object l' associated with a Port parameter object o exists and the condition λ on that connection permits this transition. The successor state then has l' as the current location and as the current loop as well. It also sets the loop index stored in memory m for the associated port object o to zero.

For the next two rules, which handle the two possible loop end cases, we define n to be the number of API ports of type $Prop_{Type}(o)$ in the configuration of partition p_a.

Transition rule 9:

$$\frac{\langle l,\lambda,l'\rangle \in T \land r \in \lambda \land l' \in Obj_{LoopEnd}(b) \land \langle \varphi,o\rangle \in P \land m(o) < n}{\langle l,\varphi,m,r,v,p_a,p_r\rangle \longrightarrow \langle \varphi,\varphi,m[m(o)+1/o],r,v,p_a,p_r\rangle}$$

This rule defines the change of state if from the current location l a control flow connection to a Loop End object l' exists, the condition λ on that connection permits this transition, and the loop index counter stored in memory m is less than n (i. e. the number of ports currently being looped over). The successor state then has the current loop start object φ as the current location and as the current loop as well. It also increments the loop index stored in memory m for the associated port object o by one. In other words, this rule starts another loop iteration.

Transition rule 10:

$$\frac{\langle l,\lambda,l'\rangle \in T \land r \in \lambda \land l' \in Obj_{LoopEnd}(b) \land \langle \varphi,o\rangle \in P \land m(o) \geq n}{\langle l,\varphi,m,r,v,p_a,p_r\rangle \longrightarrow \langle l',\bot,m,r,v,p_a,p_r\rangle}$$

This rule defines the change of state if from the current location l a control flow connection to a Loop End object l' exists, the condition λ on that connection permits this transition, and the loop index counter stored in memory m is greater than or equal to n (i. e. the number of ports currently being looped over). The successor state then has l' as the current location and $\bot$ as the current loop (i. e. no current loop). In other words, this rule terminates the current loop.

The final rule shows what happens inside a loop. Aside from other API calls as defined by previous rules, it is possible to operate on the current API port.

Transition rule 11:

$$\frac{\begin{gathered}\langle l,\lambda,l'\rangle \in T \land r \in \lambda \land l' \in Obj_{Create\_Sampling\_Port}(b) \land \langle l',o\rangle \in P \land \\ p = Port(p_a, Prop_{Type}(o), m(o))\end{gathered}}{\langle l,\varphi,m,r,v,p_a,p_r\rangle \xrightarrow[]{\substack{p_a:p_r:\texttt{Create\_Sampling\_Port}(Prop_{Name}(p), \\ Prop_{MaxMsgSize}(p),Prop_{Dir}(p),Prop_{Refresh}(p),i,r')}} \langle l',\varphi,m[i/p],r',v,p_a,p_r\rangle}$$

This rule defines the change of state if from the current location l a control flow connection to a Create_Sampling_Port object l' associated with a Port parameter object o exists and the condition λ on the control flow connection permits this transition. This transition results in a Create_Sampling_Port API call addressed to the current process p_r in the current partition p_a, with its input parameters taken from the appropriate properties of the port p from the partition's configuration, as referenced by the loop index stored in memory m for the associated Port parameter object o. It returns the output parameter i and the return value r'. The successor state then has l' as the current location, r' as the last return value, and memory m is updated to contain the returned API port ID i as value for port p.

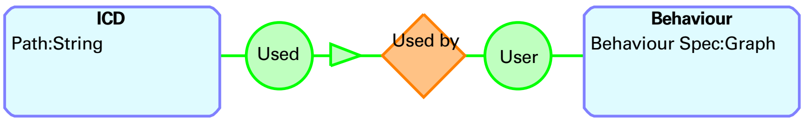


Figure 9.6: ITML-C Test Suite

The transition rules for the other port handling functions work very similar to the last one, and are therefore omitted here.

9.3 ITML-C: Configured Module Testing

The IMA Test Modelling Language variant C (ITML-C) allows the definition of Configured Module tests. Since configurations are pre-defined in this scenario, the language does not allow the definition of configurations. Therefore, ITML-C only consists of a Behaviour part and a Test Suite part. In order to implement these two parts, however, it is possible to reuse the Behaviour and Test Suite graph types defined for ITML-B in the previous section.

9.3.1 Behaviour

The Behaviour part of ITML-C consists of a graph, and its meta-model is identical to the Behaviour part defined for ITML-B.

9.3.2 Test Suite

In contrast to the Behaviour graph, the Test Suite graph type cannot be completely identical to its ITML-B counterpart. The reason is simple: While an ITML-B Test Suite refers to Configuration graphs, an ITML-C Test Suite must refer to external configurations (which have not been created within the ITML framework).

Abstract Syntax

As in ITML-B, a Behaviour object has a property referring to the Behaviour graph. But in order to connect ITML-C Behaviour graphs with existing configurations, an ITML-C Test Suite graph has an ICD object instead of a Configuration object. The ICD object represents an external ICD, referring to its location via the *Path* property. As shown in figure 9.6, a Behaviour object is always associated with one or more ICD objects. Therefore, the graph specifies for each test case that is part of the test suite the external configurations with which it is to be executed.

Static Semantics

Similar to ITML-B, each ITML-C Test Suite graph must be compliant with the following two constraints:

1. A Behaviour must be in at least 1 Used by relationship.
2. An ICD must be in at least 1 Used by relationship.

Since these constraints cannot be enforced directly by MetaEdit+, they must be checked by a model validity checker. The corresponding model validity checker can be found in appendix D.4.

9.3.3 Dynamic Semantics

In order to define the dynamic semantics of ITML-C test suites, we can completely reuse the definitions and rules from section 9.2.4. This is possible due to the fact that we only referenced the configuration (the API ports in particular) informally in our rules, so that it does not matter now whether we defined the configuration as an ITML Configuration graph or as an external ICD. Therefore, all rules from ITML-B apply to ITML-C as well, with the exception that the configuration *c* is not a MetaEdit+ graph, but an external ICD.

9.4 ITML-A: Hardware/Software Integration Testing

The IMA Test Modelling Language variant A (ITML-A) allows the definition of hardware/software integration tests. With ITML-A, a test expert can specify application behaviour and environment simulations in the form of models. The ITML-A test case generator can automatically derive test cases from these models and generate appropriate TTCN-3 test procedures.

9.4.1 System Diagram

The application behaviour is specified as a composition of functional components, each of which can consist of subcomponents. Components that do not consist of subcomponents must be specified as statechart graphs.

The environmental behaviour can be specified in the same manner as the application behaviour, i. e. via a composition of components and statechart graphs.

Both the application (SUT) and the environment (TE) appear in a system diagram, the top-level graph type of ITML-A.

Abstract Syntax

Figure 9.7 shows the abstract syntax of the system diagram. It contains one instance of an SUT object and one instance of a TE object, both of which

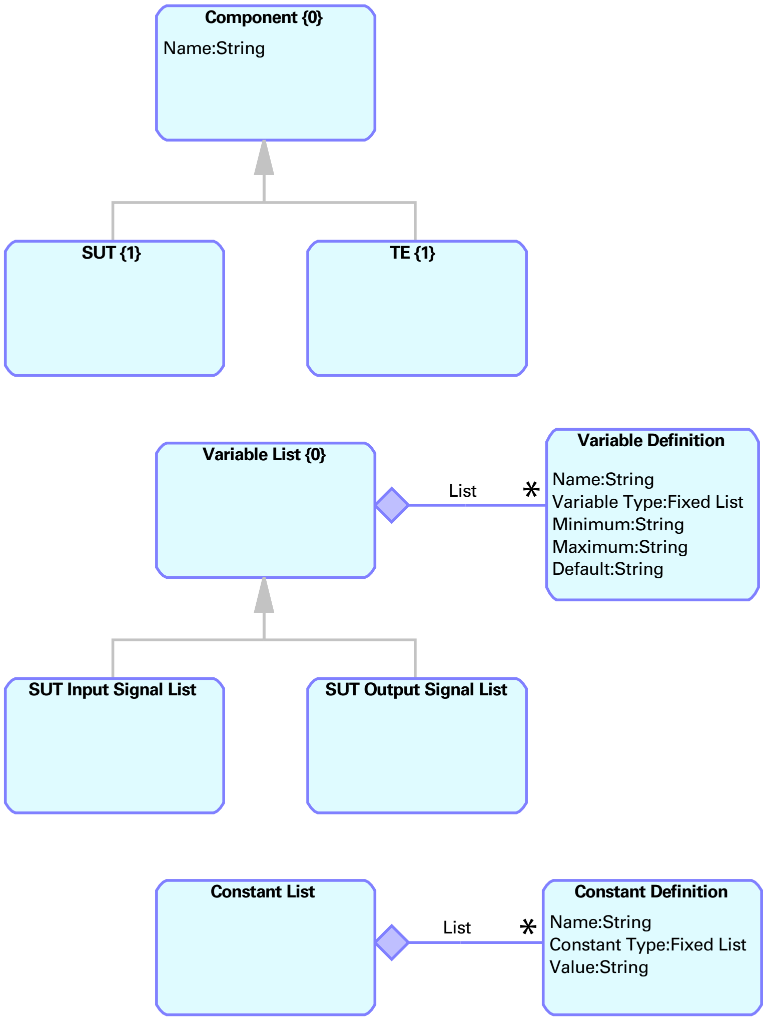


Figure 9.7: Abstract Syntax: System Diagram

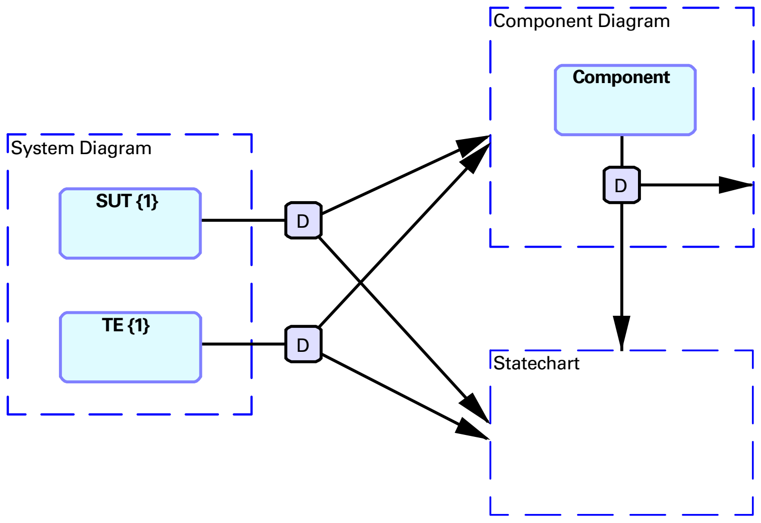


Figure 9.8: Abstract Syntax: System Diagram Decomposition

are specialisations of the Component object type (which is an abstract base type in this graph).

The graph can contain instances of SUT input signal lists and SUT output signal lists. Both are derived from the (abstract) variable list object type. Variable list objects contain a list of variable definition objects, each definition defining one variable, or in this case, one input or output signal of the SUT.

The system diagram may also contain one or more constant lists. Each list object contains constant definitions. Constants defined in the system diagram have global visibility, i. e. they can be referenced in both the SUT and the TE.

We already mentioned briefly that in ITML-A there is a hierarchy of graphs. This is also reflected in the abstract syntax. Figure 9.8 shows the decomposition relationships between object and graph types of ITML-A. In particular, both the SUT object and the TE object in a system diagram can decompose to either a component diagram or a statechart. In a component diagram (see below), each component object can decompose to either another component diagram or to a statechart. This leads to a tree of graphs, with the system diagram as the root node, component diagrams as internal nodes, and statecharts as leaf nodes.

Static Semantics

As determined during the domain analysis, a system diagram graph must adhere to the following constraints:

1. SUT may occur at most 1 time.
2. TE may occur at most 1 time.

While MetaEdit+ directly supports the previous constraints, the remaining constraints cannot be checked automatically by MetaEdit+ and must instead be checked by a model validity checker:

3. SUT must occur exactly 1 time.
4. TE must occur exactly 1 time.
5. The name of each signal and constant definition must be unique[6].
6. SUT must decompose to either a component diagram or a statechart.

Note that while the SUT must decompose, this is not required for the TE, and therefore no corresponding constraint is needed. The corresponding model validity checker can be found in appendix D.5.

9.4.2 Component Diagram

A component diagram shows the decomposition of the SUT, TE, or a component into subcomponents.

Abstract Syntax

Figure 9.9 shows the abstract syntax of the component diagram. The component diagram graph type is very similar to the system diagram. However, there are two differences. First, instead of the SUT and the TE, the diagram shows one or more component objects. Each component has a name and can decompose to either another component diagram or a statechart. And second, instead of input and output signals it is possible to declare component-local variables in variable list objects. Each variable list object has a list of variable definitions. A variable declared in a component is visible within all subcomponents of that component.

Like the system diagram, the component diagram may also contain one or more constant lists. Each list object contains constant definitions. Like variables, constants defined in the component diagram have component-level visibility, i. e. they can be referenced in all subcomponents.

Note that in contrast to the system diagram, component and variable list are not abstract but concrete object types in the component diagram.

[6]MetaEdit+ does support uniqueness constraints of object properties in a graph type, but not across multiple object types (i. e. signals and constants sharing a common namespace).

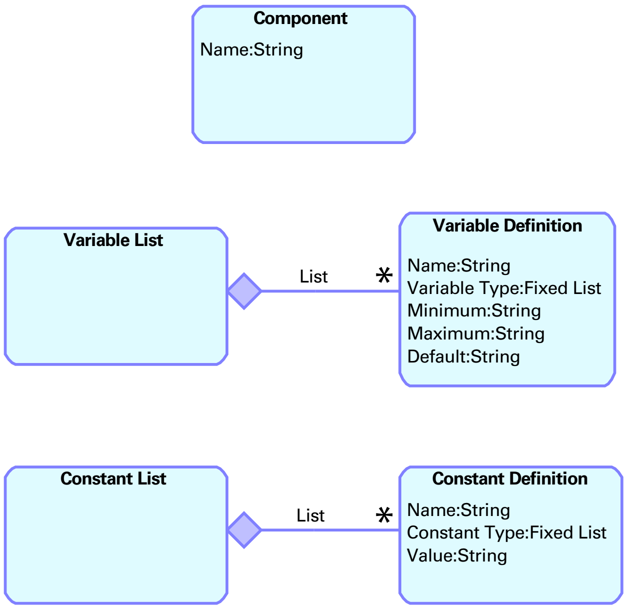


Figure 9.9: Abstract Syntax: Component Diagram

Static Semantics

As determined during the domain analysis, a component diagram graph must adhere to the following constraints:

1. Property “Name” in Component must have unique values.

While MetaEdit+ directly supports the previous constraint, the remaining constraints cannot be checked automatically by MetaEdit+ and must instead be checked by a model validity checker:

2. Component must occur at least 1 time.

3. The name of each variable and constant definition must be unique.

4. A component which is part of the SUT must decompose to either a component diagram or a statechart.

The corresponding model validity checker can be found in appendix D.6 (the last constraint is actually checked by the system diagram validator).

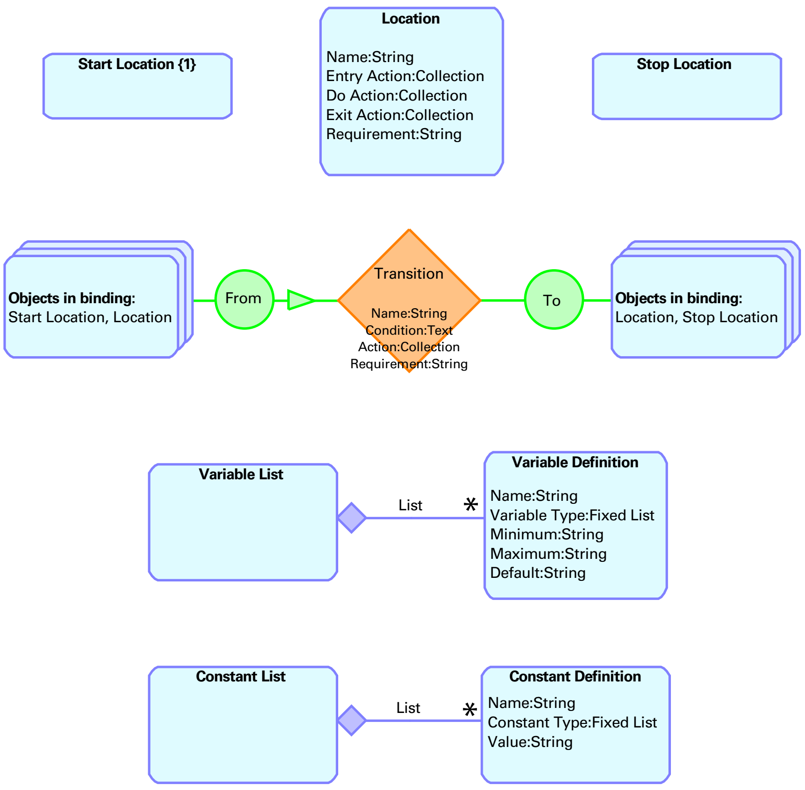


Figure 9.10: Abstract Syntax: Statechart

9.4.3 Statechart

The statechart graph type represents the lowest level of the graph hierarchy. It shows a statechart representation of the behaviour of a component which is not divisible into subcomponents.

Abstract Syntax

Figure 9.10 shows the abstract syntax of the statechart. A statechart graph consists of state objects, called *locations*. Locations have as properties: a name, a list of entry, exit, and do actions, and a requirement tag.

Location objects are connected via directed transition relationships. A transition has a label (i. e. name), a guard condition, and a list of actions as properties. Like locations, it can also have a requirement tag.

A statechart graph has exactly one initial state, called the start location.

Similar to component diagrams, a statechart can also have variable list objects and constant list objects. These variable and constant definitions have statechart-level visibility, i. e. they can be referenced only within the statechart.

Action properties consist of assignments to output signals or internal variables, while guard conditions are written as predicates on input signals or internal variables.

Static Semantics

As determined during the domain analysis, a statechart graph must adhere to the following constraints:

1. Start Location may occur at most 1 time.
2. TE may occur at most 1 time.

While MetaEdit+ directly supports the previous constraints, the remaining constraints cannot be checked automatically by MetaEdit+ and must instead be checked by a model validity checker:

3. Start Location must occur exactly 1 time.
4. The name of each variable and constant definition must be unique.

The corresponding model validity checker can be found in appendix D.7. There are, however, additional constraints that can neither be checked by MetaEdit+ nor a model validator written in MERL:

5. Non-empty guard conditions must be logical expressions.
6. Non-empty actions must be a sequence of assignments.
7. Variable, constant, and signal names occurring in expressions must be defined in the scope of the expression or a parent scope.

Guard conditions must be logical expressions which evaluate to *true* or *false*. In particular, everything that can be derived from the *logical-OR-expression* nonterminal symbol of the C language grammar (cf. [KR88, A7.15]) is allowed. Actions must be a list of assignments corresponding to the *assignment-expression* of the C language grammar (cf. [KR88, A7.17]). Additionally, *selection-statements* (cf. [KR88, A9.4]) are permitted, but their subordinate *statements* can only be *assignment-expressions* or nested *selection-statements*.

MERL generators are not adequate to check these constraints, as their text recognition capabilities do not go beyond regular expressions. Instead, conditions and actions have to be exported as strings from MetaEdit+ during model export, and the test case generator checks conditions and actions for validity by employing appropriate Bison-based parsers and afterwards resolving symbol names.

Note also that statecharts describing environmental behaviour may be non-deterministic, while statecharts modelling (SUT) application behaviour must be deterministic.

9.4.4 Dynamic Semantics

The dynamic semantics of a single statechart graph c are a variant of Harel Statechart semantics (cf. [Har87]). They can be described as follows (we reuse the auxiliary definitions from section 9.2.4 here):
Let L be the set of all locations in statechart c:

$$L = Obj_{StartLocation}(c) \cup Obj_{StopLocation}(c) \cup Obj_{Location}(c)$$

Let N be the set of transition relationship identifiers[7] in statechart c:

$$N = \{Prop_{id}(Rel_{Transition}(c, l_1, l_2)) \mid l_1, l_2 \in L\}$$

Let I be the set of defined SUT input signals, O be the set of defined SUT output signals, and V be the set of internal model variables. Then let M be a set of memory functions $m : I \cup O \cup V \rightarrow D$ which maps signal and variable symbols to their current value in their respective domain (D being the superset of all these domains).
We define a statechart transition system C as a 9-tuple:

$$C = \langle S, N, T, G_T, A_T, A_{En}, A_{Do}, A_{Ex}, s_0 \rangle$$

where

$S = L \times M$	state space,
$T \subseteq L \times N \times L$	transition relation,
$G_T : N \times M \rightarrow \{\text{true}, \text{false}\}$	guard condition function
$A_T : N \times M \rightarrow M$	transition action function
$A_{En} : L \times M \rightarrow M$	entry action function
$A_{Do} : L \times M \rightarrow M$	do action function
$A_{Ex} : L \times M \rightarrow M$	exit action function
$s_0 \in S$	initial state.

The transition relation T contains tuples representing transitions in the statechart graph c:

$$\langle l, n, l' \rangle \in T \Leftrightarrow l' \in RelO_{Transition}(c, l) \cap Role_{To}(c, l) \wedge \\ n = Prop_{id}(Rel_{Transition}(c, l, l'))$$

[7]The IDs are allocated internally by MetaEdit+ and guaranteed to be unique.

A transition tuple $\langle l, n, l' \rangle$ is element of the transition relation iff the statechart graph contains a transition relationship from l to l' and that relationship has identifier n.

The guard function G_T is defined so that it returns true iff memory function m satisfies the guard condition of the transition with identifier n:

$$G_T(n,m) = \begin{cases} \text{true} & \text{if } m \vDash Prop_{Condition}(r) \;\wedge\; Prop_{id}(r) = n \\ \text{false} & \text{otherwise} \end{cases}$$

Note that, as a special case, every memory function satisfies the empty guard condition.

The transition action function A_T is defined so that it returns a copy of the input memory m with all actions of the transition with identifier n applied to it. For example, if a transition relationship r has $Prop_{Action}(r) = \{\text{output1} = 5\}$, then

$$A_T(Prop_{id}(r), m) = m[5/\text{output1}]$$

The entry action, do action, and exit action functions (A_{En}, A_{Do}, A_{Ex}) are defined correspondingly, but with properties $Prop_{EntryAction}(l)$, $Prop_{DoAction}(l)$, and $Prop_{ExitAction}(l)$ of location l.

Initial State

For a given statechart graph c, the initial state is the tuple

$$s_0 = \langle l_0, m_0 \rangle$$

where

$$\begin{aligned} l_0 &\in Obj_{StartLocation}(c) && \text{start location,} \\ m_0(o) &= 0 \;\; \forall\, o \in I \cup O \cup V && \text{zero-initialised memory.} \end{aligned}$$

In particular, the start location l_0 is set to the only[8] Start Location object in the graph. The definition of m_0 assigns the value 0 to all input signals, output signals, and model variables.

Transition Rules

A set of transition rules defines how the state of the transition system changes. If a transition rule is possible, it must be applied. If more than one transition rule can be applied in the current state, the rule with the lowest number must be applied. We need to distinguish three different situations which lead to a change in the state of the statechart transition system.

[8] as per static semantics rule no. 3

Transition rule 1:

$$\frac{i \in I \land n \in D_i}{\langle l, m \rangle \longrightarrow \langle l, m[n/i] \rangle}$$

The first situation is the change of an external input: Any SUT input i can change to an arbitrary value n (from the domain of i) without restriction. This is reflected in the update on memory m.

Transition rule 2:

$$\frac{\langle l, n, l' \rangle \in T \land G_T(n, m)}{\langle l, m \rangle \longrightarrow \langle l', A_{En}(l', A_T(n, A_{Ex}(l, m))) \rangle}$$

In the second situation a guard condition is satisfied and allows the transition into another location. A transition with identifier n exists from the current location l to another location l'. The guard function for transition n is satisfied with current memory state m. This leads to a transition into successor location l', thereby applying the exit action of location l, the action of transition n, and finally the entry action of location l' to memory m. (Note: In statecharts which are located in the test environment, the application of this rule is not mandatory (in contrast to statecharts in the SUT). This means that statechart transitions in the test environment are not urgent and can be taken indeterministically.)

Transition rule 3:

$$\frac{}{\langle l, m \rangle \longrightarrow \langle l, A_{Do}(l, m[m(t) + \varepsilon/t]) \rangle \mid t \in V \land timer(t)}$$

In the third situation no transition to another location is possible. In this case, time passes (all model variables t which are timers are incremented by some value $\varepsilon > 0$), and the do action of the current location l is applied to the memory m.

CHAPTER 10

Test Generation

This chapter gives a detailed description of the ITML Test Case Generators. Their task is the generation of executable TTCN-3 code from ITML models.

As presented in section 3.2.4, the MetaEdit+ framework provides a special generator language, called MERL, in which generators can be written. While MERL (itself being a DSL) is well suited for navigating graphical GOPPRR-based models and creating output text files, it lacks some more general-purpose programming concepts like custom data types, scoped variables, and custom functions. Consequently, depending on the complexity of the task demanded from the generator, it can be deemed impractical to implement the generator completely in MERL. A reasonable solution to this problem is to have a very simple generator, written in MERL, which generates a file containing an intermediate representation of the model. Then this intermediate file can be processed by a more complicated generator written, for example, in a general-purpose language like C++ or Python.

10.1 MERL Intermediate Format Generators

As the task of generating executable TTCN-3 code from ITML models is rather complex and involves more inputs than just the model—also an external configuration in ICD format—it is not feasible to implement the whole generator in MERL. Instead, the ITML generators written in MERL produce files containing intermediate XML representations of the different model graphs. These generators are called MERL *Intermediate Format Generators* (IFGs). For each graph type there is a dedicated IFG. Their definitions can be found in appendix E as MERL source code. As can be seen in figure 10.1, the IFGs produce XML representations of the model inputs. XML schemas have been defined for these XML file formats. The appro-

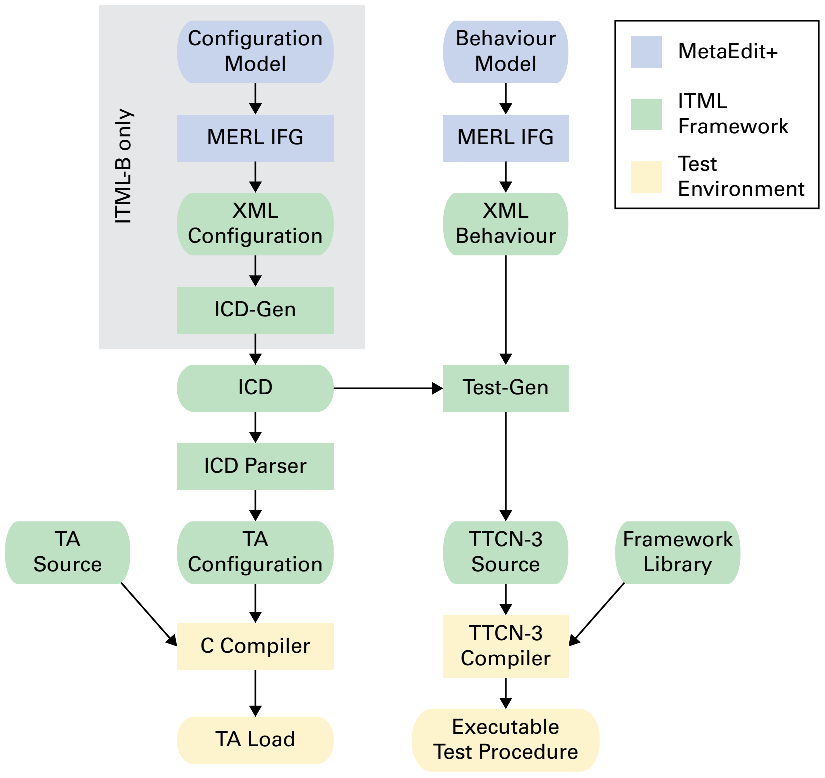


Figure 10.1: ITML-B and ITML-C generation process

priate XML Schema [W3C04] definitions can be found in appendix F. The intermediate XML files contain information about what objects are present in a graph, what properties the objects have, and by what relationships they are connected. The exact graphical layout of the graph elements, however, is not stored inside the XML files, as this information is not relevant for the generation process.

10.2 Generation of ITML-B and ITML-C Tests

Figure 10.1 provides an overview of the generation process within the ITML framework. It shows generators (rectangles) and artefacts (rounded boxes), which are the generator inputs and outputs.

10.2.1 ICD Generator

The ICD Generator (ICD-Gen) is employed within the ITML framework to create actual ICDs from Configuration XML files (produced by the IFG). Note that this step is only performed for *Bare Module* testing in ITML-B, whereas ITML-C works with externally supplied configurations. The generator reads an XML intermediate configuration file and produces a set of CSV files. For each partition defined in the configuration model, two files are generated. They contain configuration information as described in section 2.3, derived from the XML configuration file contents. The files are generated so that they are suitable for processing with the module-specific configuration and load generation toolchain as provided by the module supplier[1].

The ICD Generator is a command-line application written in C++. It uses the XSD Data Binding compiler [CST] to parse the XML input file and has its own CSV file export routines. Due to its size the complete source code is not reproduced here.

10.2.2 ICD Parser

After generating the ICD as described in the previous section, or when using an external ICD, it has to be processed further. The test generator itself requires it as input (cf. next section), but it is also required in order to generate the partition-specific part of the universal Test Agent (cf. chapter 7). While the agent main loop and protocol handling code are completely generic, the agent requires two data tables containing information about the API ports defined within the partition and the data signals which may be defined within API port messages.

To that end, the ICD parser is used to generate a C source file containing two appropriate data structures. The ICD parser is a command-line application written in Perl. It is based on a previous parser which the author developed during his work in the VICTORIA research project.

The resulting TA configuration C source file can now be combined with the rest of the TA source code and processed by the compiler, linker, and load generation tools provided by the module supplier. The generated load is then ready for deployment on the SUT module.

10.2.3 TTCN-3 Code Generator

As shown in figure 10.1, there is a second path of generation, i. e. the processing of the Behaviour. After having employed the Behaviour IFG in order to obtain a behaviour XML file, it must be processed by the TTCN-3 Test Generator (Test-Gen). This generator takes two inputs: The aforementioned behaviour XML file, but also the ICD (either generated or supplied externally) associated with the behaviour model (as defined in the test suite).

[1]The manufacturer of the IMA module

The test generator is the most complex of the generators described here because it has to generate TTCN-3 code that results in API calls conforming to the dynamic behaviour of its input model and configuration as specified by the behaviour transition system defined in section 9.2.4.

The TTCN-3 Test Generator is a command-line application written in C++. It uses the XSD Data Binding compiler [CST] to parse the XML input file and has its own CSV file import routines. Due to its size the complete source code is not reproduced here.

Goto Considered Harmful

A rather straight-forward method of generating code from a flowchart is to create a blocks of statements prefixed by a unique label for each node and to generate a `goto` statement for each edge between nodes.

```
...
// node 1
label b1;
stmt1;
stmt2;
if (cond1) {
  goto b2;
}
goto b3;

// node 2
label b2;
stmt3;
stmt4;
goto b3;
...
```

Unfortunately, it was discovered during development of the generator that the TTCN-3 compiler in TTworkbench only supports a limited number of labels and gotos per file. Exceeding this limit results in syntactical errors in the generated Java source. Note that the Java language has no direct support for `goto` (cf. [ORC]), so the TTCN-3 compiler uses a construct of loops and switch statements to simulate it.

A practical work-around for this problem was to perform this simulation of `goto`s directly in TTCN-3 instead of using `goto`s in the TTCN-3 code. This can easily be achieved by declaring a variable `node` to be used as the current label and then generating the statements for each node inside a sequence of `if`-blocks. All these `if`-blocks are encapsulated within a `while`-loop. Now, instead of performing a `goto` statement, the `node` variable is set to the number of the next node, and a `continue` statement is used to restart the `while` loop.

```
int node := 1;
while (node != 0) {
  // node 1
  if (node == 1) {
    stmt1;
    stmt2;
    if (cond1) {
      node := 2;
      continue;
    }
    node := 3;
    continue;
  }

  // node 2
  if (node == 2) {
    stmt1;
    stmt2;
    node := 3;
    continue;
  }
  ...
}
```

10.3 Generation of ITML-A Tests

The ITML-A generation process is shown in figure 10.2. While similar, it is not identical to the process shown before. The following steps make use of *rtt-tcgen*, the model-based testing environment developed by the AGBS research group[2] at the University of Bremen.

Like before, a MERL IFG is used to produce an intermediate XML representation of the system model. The corresponding generator can be found in appendix E.5. The XML schema for the intermediate format is presented in appendix F.5.

Before the XML model export can be processed by the actual test case generator, it must be processed by two additional tools.

conftool creates a configuration file for the test case generator. It contains a list of all components and the transitions inside their statecharts. The test engineer can use this file to set focus points (transitions that must be covered) or mark transitions as unreachable or robustness transitions.

[2]`http://www.informatik.uni-bremen.de/agbs/`

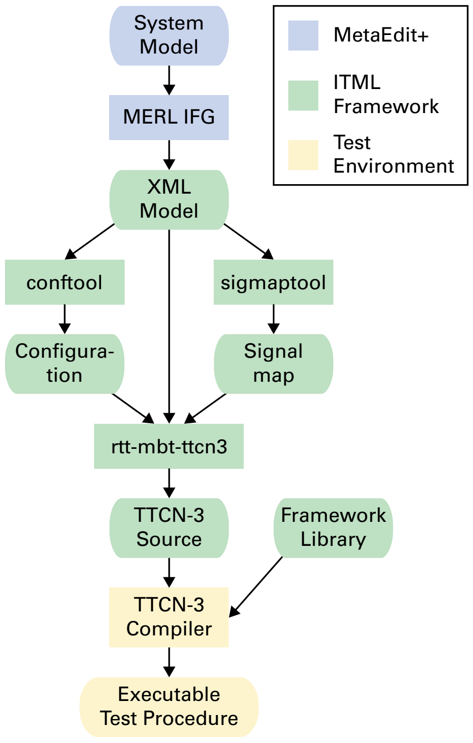


Figure 10.2: ITML-A generation process

sigmaptool creates a table containing all signals (i. e. SUT inputs and outputs) defined in the system model. The value ranges as defined in the model can be adjusted by the user. Tolerances and signal delays/latencies can be specified as well.

The files created by the aforementioned tools are in CSV format. Their contents can be adjusted, if necessary, by the test engineer before they are processed together with the XML model export by the test case generator.

10.3.1 Test Case Generator

The actual TTCN-3 test case generator tool is based on existing test case generators in the rtt-tcgen framework. The executable is called **rtt-mbt-ttcn3**. It reads the system XML model, the configuration, and the signal map as inputs and creates a TTCN-3 source file as output.

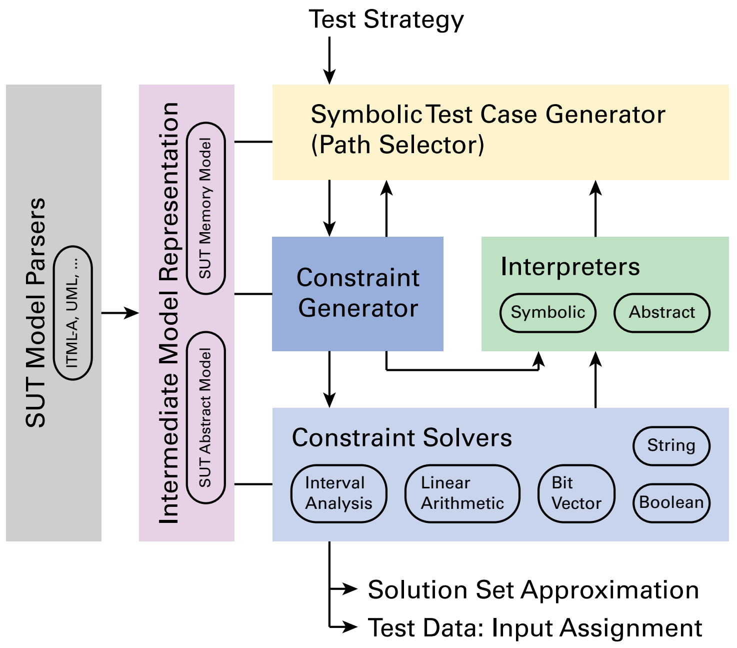


Figure 10.3: Test Case Generator architecture

Model Parser

The generator first reads the configuration file. Then it tries to parse the model. The generator has several parser back-ends, each of which is applied in turn until one successfully parses the model file. Apart from ITML there are parser back-ends available for model exports from other modelling tools like Artisan, Enterprise Architect, or Rhapsody. The purpose of the parser back-ends is to transform the contents of the model input file into the intermediate model representation (IMR) structure used internally in the generator. Compared to other model export formats, the ITML intermediate XML format is very simple, which makes the parsing and transformation step very straightforward.

Generator

While **rtt-mbt-ttcn3** makes use of the test case generation functionality of *rtt-tcgen*, the internal workflow of the generator is not a part of this thesis. It is described in detail in [Pel13], and only a short summary will be given here.

Figure 10.3 shows the internal structure of the generator. From the intermediate model representation a transition relation and initial state are generated. According to the configured test strategy, model paths are selected. Such paths, expressed as LTL formulae, are transformed to so-called *symbolic test cases* (cf. [Pel13, p. 14]). Now an SMT solver (cf. [RT06]) is employed to find a solution in the form of a valid trace from the initial state to the desired goal (test objective) in compliance with the model's transition relation. An abstract interpreter supports the solver by determining required transition steps and restricting variable ranges. If a solution exists, the result is a trace of timestamps and concrete input data for the test procedure.

TTCN-3 Output

The results of the previous step are then processed by the TTCN-3 output back-end. The back-end's task is the creation of a TTCN-3 source file containing a test procedure which covers the generated test cases. The generated TTCN-3 file has the following structure:

- port type declarations
- MTC component
  - ports
  - I/O variables
  - state variables
  - cycle timer
  - global time-tick variable
  - oracle functions
  - stimulator function
  - input function
  - output function
- testcase
  - main loop

For each input and output signal from the signal map a TTCN-3 port type declaration with appropriate data type and direction (TTCN-3 output port for SUT inputs, TTCN-3 input port for SUT outputs) is generated. The MTC component contains a corresponding port definition for each signal, so that it can communicate with the SUT via the SUT adapter (cf. section 5.4). For each port there is also a component variable (called *input variable* or *output variable*).

The TTCN-3 testcase is generated to start a cycle timer and execute an endless loop consisting of four steps:

1. call input function

2. call oracle functions
3. call stimulator function
4. call output function

The input function consists of an `alt` statement over all input ports (i. e. SUT output signals) plus the cycle timer. If a value can be read from an input port, it is stored in the corresponding input variable. Then the `alt` statement is repeated. The statement is only left when the cycle timer expires, in which case the timer is restarted, the cycle time is added to the global time-tick variable, and the main loop proceeds to the oracle functions.

The TTCN-3 output back-end generates a set of *oracle* functions. It traverses the IMR to find all leaf components. From each component's statechart it creates an abstract syntax tree (AST), which is then translated into TTCN-3 code as the body of the component's oracle function. A branch is added to the AST of the function for each location of the statechart. If a location contains do actions, they are added to the AST branch. From each location all outgoing transitions are considered. For each outgoing transition an *if* statement is added to the AST branch with the transition's guard as condition. The body of the *if* statement contains the transition's actions and the entry actions of the destination location as well as an assignment of the new location to the oracle's state variable in the MTC. In other words, running the oracle functions creates the same variable and signal assignments as the SUT should do. This allows the oracle functions to run back-to-back with the SUT and perform a comparison of the expected (as calculated by the oracle) and actual SUT output signals. The oracle function contains `log` statements printing the respective requirements to the test log, and `setverdict` statements to log the result of the comparison of actual and expected values.

The purpose of the stimulator function is to provide stimuli to the SUT in order to cover the test cases determined by the generator. The stimulator function implements a simple state machine. Its current state is stored in a component variable of the MTC. Depending on the global time-tick, the function makes assignments to output variables. The TTCN-3 output back-end traverses the computation tree built by the generator to determine the points in time and examines the internal memory model to determine the assignments on output variables (i. e. SUT inputs) to be made. When the stimulator function reaches the end of its state machine, it terminates the test procedure by executing the `stop` statement.

Finally, the output function is called. This function checks if the values of output variables have changed during this cycle. For each changed variable, its new value is sent via the corresponding port to the SUT. Then the main loop starts again with the input function.

Test Execution

The generated TTCN-3 source file contains the complete test procedure implementation in order to cover the test cases generated by the test case generator from the model. The test engineer can now compile the test procedure in the TTCN-3 test execution environment (cf. chapter 8). Note, however, that for functional testing the TACP sub-SA must be removed from the SUT adapter (because there are no agents involved). Instead, the adapter must map the defined signal ports to the appropriate hardware interfaces of the module (by consulting the list of signals in the actual module ICD as specified by the module integrator).

PART III

Evaluation

CHAPTER 11

Usage in the SCARLETT Project

As already mentioned earlier, the ITML framework is based in part on research performed within the European research project SCARLETT (cf. section 2.5).

During work package WP2.4 (Toolset & Simulators Development) a test case generation and execution process was defined (project-internal deliverable SCA-WP2.4-ADE_TECH-D_2.4-01-ICI4). Based on this process, specifications for the framework components (DSL, Test Agent, generator tools) were elaborated (SCA-WP2.4-ADE_TECH-D_2.4-02-ICI7). Mock-ups and working prototypes of the framework components were developed in order to complete the work package.

Afterwards, the developed components were used in the demonstration phase of the project. In work package WP4.2 (I/O Intensive Capability Demonstration), a demonstration platform was built to showcase IMA2G technology in I/O-intensive setups.

11.1 I/O-intensive Scenarios

Several scenarios were defined in WP4.2 in order to perform I/O-intensive tests on the available IMA2G hardware. In these scenarios the Test Agent was running on a CPM, providing stimuli to various interface hardware.

In order to produce the load for the CPM, the ICD (provided by the platform integrator) was processed with the TA configuration parser. Then the resulting configuration tables and the generic Test Agent source code were compiled with the Wind River C compiler. The resulting image was downloaded to the module with an ARINC 615A data loader (cf. [Aer07a]).

The test environment consisted of TTworkbench as a TTCN-3 environment and an ADS2 system for access to hardware interfaces. The Wireshark plug-in was utilised as well to analyse communication between the test environment and the Test Agent. A prototype version of the test generator was used to generate appropriate TTCN-3 test procedures for the individual scenarios. These test procedures were compiled together with the Proxy SA, the TACP SA, and an ADS2 SA and then executed within TTworkbench.

11.1.1 Scenario 1

The first scenario consists of a CPM, an AFDX switch, and two RDCs. The RDCs are connected to the CPM via the switch. The two RDCs are connected to each other: Each discrete output of RDC 1 is wired to a discrete input of RDC 2 and vice versa. Similarly, analogue outputs are wired to analogue inputs. The CAN buses of the RDCs are linked as well.

The Test Agent is employed to command the RDCs to toggle each output discrete and then read back the corresponding input discrete from the other RDC. It does the same by applying different voltages to the analogue outputs and reading the active voltage back from the connected analogue input. CAN bus messages are sent in both directions and received from the respective other endpoint.

11.1.2 Scenario 2

The second scenario consists of a CPM, an AFDX switch, an REU, and a stepper motor. The REU is connected to the CPM via the AFDX switch, while the control interface of the stepper motor is connected to a special interface of the REU. The stepper motor also has a read-back interface, which is connected to the test environment.

In this scenario the test procedure commands the Test Agent to send positioning commands to the stepper motor. The test procedure checks the correct rotation by reading the current angle from the test environment (ADS2 interface).

11.1.3 Scenario 3

The third scenario consists of a CPM, an AFDX switch, an RDC, and a DC bus monitor. The RDC is connected to the CPM via the AFDX switch. Two DC bus interfaces of the RDC are connected with each other (loop-back), with the DC bus monitor in between.

The Test Agent is used to repeatedly send and receive messages over the DC bus loop-back connection. In parallel, the DC bus monitor is employed to simulate noise and to add attenuation to the bus connection (this must be done manually as the monitor has no direct interface to the test environment).

11.1.4 Scenario 4

The fourth scenario consists of a CPM, an AFDX switch, an REU, and RDC, and an RPC. The REU is connected to the CPM via the AFDX switch. The RDC and the RPC are connected to the REU via FlexRay in a star topology (i. e. the REU is used as a FlexRay gateway). Discrete inputs and outputs of the RDC and RPC are connected to the test environment.

In this scenario the test procedure commands the Test Agent to send commands to the RDC and RPC via the REU in order to toggle the outputs of the RDC and the RPC. The test procedure checks the appropriate inputs of its test environment. Conversely, it toggles discrete outputs of the test environment connected to the discrete inputs of the RDC and commands the Test Agent to read back those inputs from the RDC.

11.1.5 Models

During the implementation of the framework and generators, they were tested with more generic example test models (i. e. not specific to the SCARLETT I/O-intensive demonstrator). These models are shown and described in appendix A.

CHAPTER 12

Comparison with Other Tools and Approaches

In order to give an evaluation of ITML, it is important to take a look at the intended goals. The key aspects which must be considered are:

Learning curve: A domain expert who has not used the framework before should be able to start using it without having to invest time in learning and/or memorising additional languages and/or concepts.

Intuitiveness: The meaning of the framework or language elements should be intuitively clear to a domain expert.

Efficiency: The use of the framework should provide an increase in efficiency, i. e. reduce work effort and ultimately save money.

Maintainability: It should be easily possible to adapt previously created tests to changes in requirements or in the test environment. A person different from the original author should be able to understand existing tests and make changes.

12.1 TTCN-3

The first and very interesting question is: How does ITML compare to "bare" TTCN-3? Is it worth having another layer on top of a language designed specifically for testing?

A domain expert who has never written any TTCN-3 code will first have to learn several concepts of the language. And while this is not overly difficult for someone who has at least some experience in programming, there are several unique aspects of which a deeper understanding is required in order to successfully employ TTCN-3 (cf. chapter 5, e. g. templates, ports, codecs). In contrast, the elements of ITML have a counterpart in the application domain, and therefore the domain expert is already familiar with them.

It is also important to note that ITML works on a higher abstraction level than TTCN-3. This becomes particularly apparent when considering API calls in ITML-B. To perform an API call in ITML-B requires in its simplest form a single language element. In TTCN-3 the programmer will have to take care of creating a command message, handling the TACP protocol, and receiving the response message. Note, however, that in both cases (ITML and TTCN-3), an appropriate SUT adapter provides abstraction from the specifics of accessing the TE interface hardware (and therefore allowing changes in the test environment).

The graphical representation of ITML test models makes it possible to understand them easily. The graph editor provided by MetaEdit+ is well suited to make modifications to existing models with manageable effort.

It is also interesting to note that the TTCN-3 standard defines a graphical representation of TTCN-3 test procedures, called GFT (cf. [ETS07]). It provides graphical representations for the TTCN-3 core language elements. The standard defines a mapping between the language and the GFT. This, however, means that GFT does not provide a higher level of abstraction as ITML does.

Two very important disadvantages of TTCN-3 must not be ignored: The first is its lack of support for the sampling port concept (cf. section 2.2.2). This, however, can be remedied by making sure that re-transmissions occur on SUT adapter or hardware level as well as storing the last received value of a port in a variable (also note [SP13] for an interesting alternative concept).

The second aspect is that while TTCN-3 was designed for testing, it was originally not designed for real-time testing. This becomes readily apparent in the implementation of oracles and test stimulators in section 10.3.1: TTCN-3 does not provide the concept of *clocks*, so they have to be simulated with timers and counter variables. This leads to clock drift and a multitude of "timeouts" in the TTCN-3 test logs. It should be noted, however, that several extensions to the TTCN-3 standard have been proposed in order to make it real-time-capable (cf. [SMD+08]).

12.2 CSP

As already mentioned in the introduction, in an earlier research project a bare module test suite had been developed. The test specifications were written in CSP (cf. [Hoa78, Sch00]).

Compared to ITML, the use of CSP poses a significantly higher obstacle for a domain expert, as knowledge of the domain alone is not sufficient. In order to understand or even write new specifications, the test engineer requires a deep understanding of the syntax and semantics of CSP. Also, the test engineer receives no tool support[1] during the development of test specifications and has to rely on "trial and error" techniques (i. e. writing CSP code, trying to compile it, fixing compilation errors).

In contrast, MetaEdit+ provides a much more intuitive user interface for ITML: In the graph editor, the user can see what types of objects and relationships are available. There is no need to remember names or keywords for language features or grammar rules. The automatic compliance with the rules of the metamodel enforced by the framework guarantees a syntactically correct test specification model.

The fact that CSP specifications are difficult to understand greatly affects their maintainability. Before making changes to an existing CSP specification it takes additional effort to analyse the current behaviour for a test engineer unfamiliar with the existing test suite.

There is, however, an important similarity between ITML and CSP: Both operate on an abstract level, hiding details of hardware interface access from the test specification. In fact, for CSP tests one or more *interface modules* are required, which have to provide a refinement from the abstract CSP events to physical signals on hardware interfaces to the SUT and vice versa.

A great advantage of the ITML framework is, of course, the fully automatic generation of test cases from application models with ITML-A. In contrast, test cases had to be derived and translated into appropriate CSP specifications completely by hand.

12.3 UML Testing Profile

The third and final object of comparison is the UML Testing Profile (UTP, cf. [OMG13]). UTP is a UML profile, extending UML to allow the definition and specification of testing-related artefacts (either stand-alone or within a "regular" UML model of an application or system).

UTP defines concepts to describe the structure of the test setup (SUT, TestComponent, TestContext). Test behaviour can be modelled with UML behaviour diagrams (e. g. activity diagram or state machine). Test behaviour consists of actions like SUT stimulation, logging, or test verdict calculation (validation actions). There are special concepts provided in order to specify test data: data pools, partitions, and selectors. Furthermore, the concept of test objectives supports the planning and scheduling of tests.

It quickly becomes apparent that, in terms of generalisation, UTP is the exact opposite of ITML. UTP is a general-purpose modelling language, while ITML is domain-specific, even though they are both designed for testing. In order to work with UTP, the test engineer must know and understand the special concepts of UTP in addition to the application domain. It can,

[1]apart from simple syntax colouring

however, be argued that the learning curve should be relatively shallow, since these concepts are all testing-related.

UTP itself is not a complete framework, but only a language. Additional tool support for the modelling as well as for the translation of test case models into executable test procedures is required. Therefore, an appropriate set of tools must be chosen, and the test engineer must be familiar with it. Note that there is no direct support for the automatic generation of test cases like in ITML-A. Instead, test cases are modelled explicitly, like in ITML-B/C.

Like ITML, the fact that UTP modelling is independent from specific test environment hardware increases its maintainability. The fact that UML is widely known also increases the chances of a person different from the original modeller to understand and adapt existing models.

For a more detailed introduction to and discussion of UTP topics, refer to [BDG+08].

CHAPTER 13

Conclusion

The goal of this thesis was the design and development of a framework for model-based testing of Integrated Modular Avionics systems. The key approach to this goal was the utilisation of domain-specific modelling techniques. This approach was motivated by the wish to give IMA domain experts a state-of-the-art tool that can be used with a minimum of learning effort and allows a more productive way of working compared to manually developing test procedures.

13.1 Results

The main contributions provided by this thesis are described in detail in part II. These contributions together make up the model-based testing framework.

The domain-specific modelling language ITML has been defined in order to provide a means to specify tests and desired system behaviour. Due to its domain-specific nature its concepts are easily understood by IMA domain experts. Its level of abstraction makes it possible to work directly on the *problem* level instead of having to cope with the technical details of a specific test environment. ITML comes in three variants, ITML-A, ITML-B, and ITML-C, in order to adequately cover different levels of testing. For all variants formal definitions of abstract syntax as well as static and dynamic semantics are provided. An implementation of all language variants is provided in the form of meta-models in the DSM tool MetaEdit+. This tool provides an appropriate modelling front-end to be used by IMA domain experts. Furthermore, generators have been implemented that automatically produce executable TTCN-3 test procedures based on the models designed by the domain expert.

The framework also provides a suitable TTCN-3-based test environment. This includes appropriate system adapters and codecs which can be used, for example, in TTworkbench. The choice of TTCN-3 has the advantage of having test procedures which are independent from the underlying concrete test system (due to the interfaces defined by the TTCN-3 standard).

To complement the TTCN-3 test environment, the framework provides an implementation of a flexible and universal Test Agent. The agent is ARINC 653-compatible and is therefore suited for loading onto the target module under test. The agent can execute API calls on behalf of the running test procedure. As a means of controlling the agent, a control protocol has been specified, and the framework comes with a suitable protocol implemention in the test environment.

The appendix of this thesis contains a series of example ITML models. When comparing the graphical model representations with the TTCN-3 code generated from them, it becomes easily apparent that an IMA domain expert can work more effortlessly and efficiently by creating ITML models compared to manually writing test procedures and that these models have a higher comprehensibility compared to TTCN-3 code.

13.2 Future Work

For future work, two interesting points remain: One important aspect of the ITML framework is usability. In order to further substantiate the conclusions drawn concerning framework usability in chapter 12, a case study with IMA domain experts should be conducted. This could be performed according to the evaluation techniques described, for example, in [GP96].

The other point concerns the user interface. Currently, several separate tools have to be employed in order to follow the process workflow detailed in chapter 6. From a usability perspective, it would be preferrable to have a unified user interface that provides all functionality, from modelling to test execution, inside a single front-end application. The Eclipse framework, for instance, provides a suitable platform into which all the different tool components required for IMA testing could be integrated.

PART IV

Appendices

APPENDIX A

Example Models

This appendix shows several example models, both ITML-B and ITML-A. It also shows generated intermediate XML files as well as generated TTCN-3 code.

A.1 ITML-B

Figure A.1 shows an ITML-B configuration model. It consists of a CPM containing two partitions. Both partitions have appropriate scheduling and RAM settings. Partition 9 has a group of AFDX input ports, while partition 10 has a group of AFDX output ports. Both partitions are connected via a group of RAM ports.

Figure A.2 shows an ITML-B behaviour model. It shows how the blackboard API (cf. section 2.2.2) can be tested: In the init process, a blackboard and two processes are created. Then the partition is switched into normal mode. The first process writes a message to the blackboard by making a `DISPLAY_BLACKBOARD` API call, then the second process tries to read the message via `READ_BLACKBOARD`. If the call succeeds, the test verdict is set to *passed*, otherwise the test has *failed*.

Figure A.3 shows an example model for port handling. The init process opens all AFDX sampling output ports of the partition in a loop. Then it creates a process and sets the partition to normal mode. The created process then iterates over all previously created ports and writes a message to each port by calling `WRITE_SAMPLING_MESSAGE`. Then a *read complement* is used to receive the message (in the test environment). If writing or reading fails, the test verdict is set to *failed*, otherwise the test is *passed*.

Figure A.4 is very similar to the previous model. The test starts by creating all RAM sampling input ports in partition 10 and creating a process,

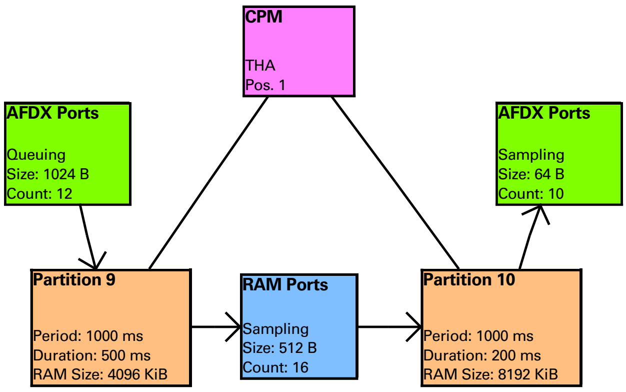


Figure A.1: Configuration model

followed by creating all RAM sampling output ports and a process in partition 9. Each partition is switched to normal mode afterwards. Then the process in partition 9 iterates over the previously opened ports and writes a message to each port. The following *read complement* will cause the process in the other partition to perform a `READ_SAMPLING_MESSAGE` on the appropriate input RAM sampling port (cf. RAM port configuration in figure A.1). If all messages are successfully received, the test is *passed*, otherwise the verdict is set to *failed*.

Appendix A.1.1 shows the intermediate XML representation that was created from the model in figure A.2 by the generator shown in appendix E.2. It contains four sections. The first two, `resources` and `nodes`, contain XML representations of the objects shown in the behaviour model (i. e. parameter objects like blackboard and processes in the resources section, API calls in the nodes section). The other two sections, `parameters` and `controlflow`, contain XML representations of the relationships between the objects in the model, i. e. parameter assignments to API calls and control flow edges.

Appendix A.1.2 finally shows the TTCN-3 test procedure generated by the ITML-B test generator from the intermediate XML representation in appendix A.1.1. The individual nodes are translated into procedure calls to the TACP handler. The function `run_Blackboard` implements the behaviour of the flowchart as described in section 10.2.3.

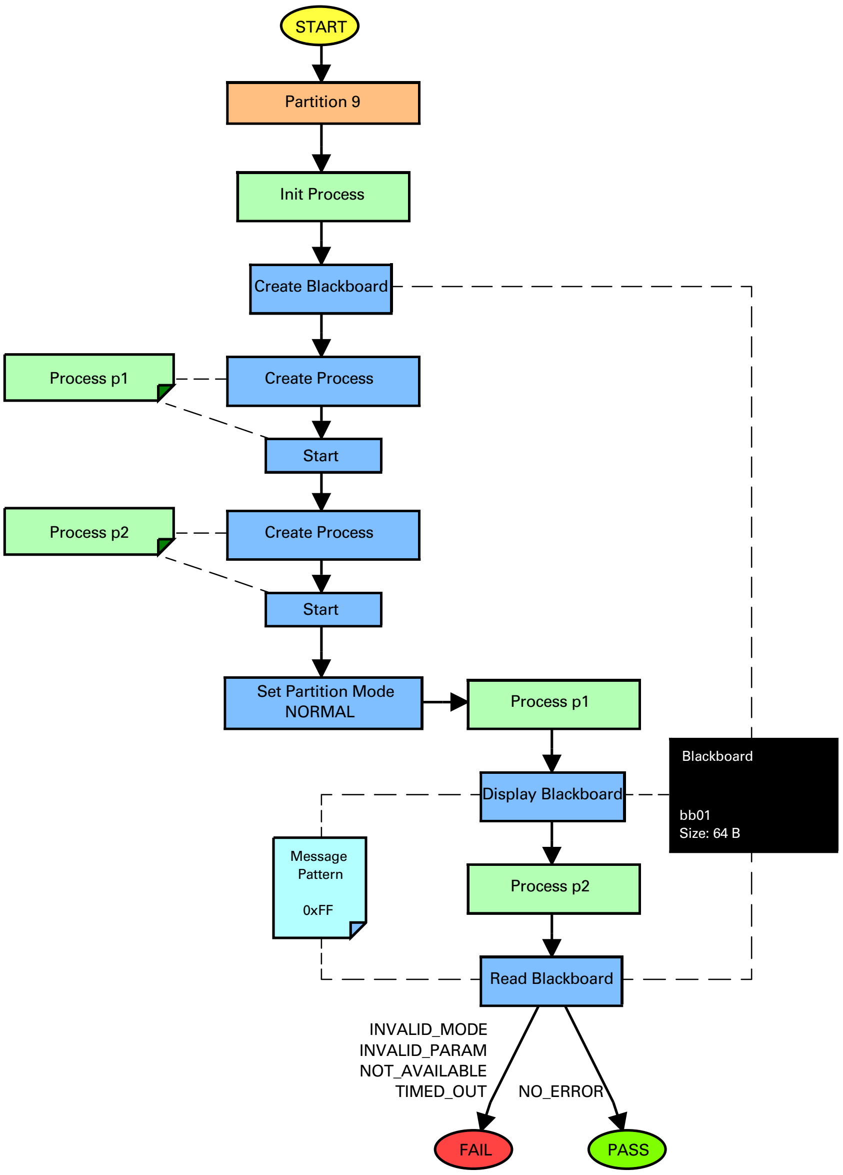


Figure A.2: Blackboard API test model

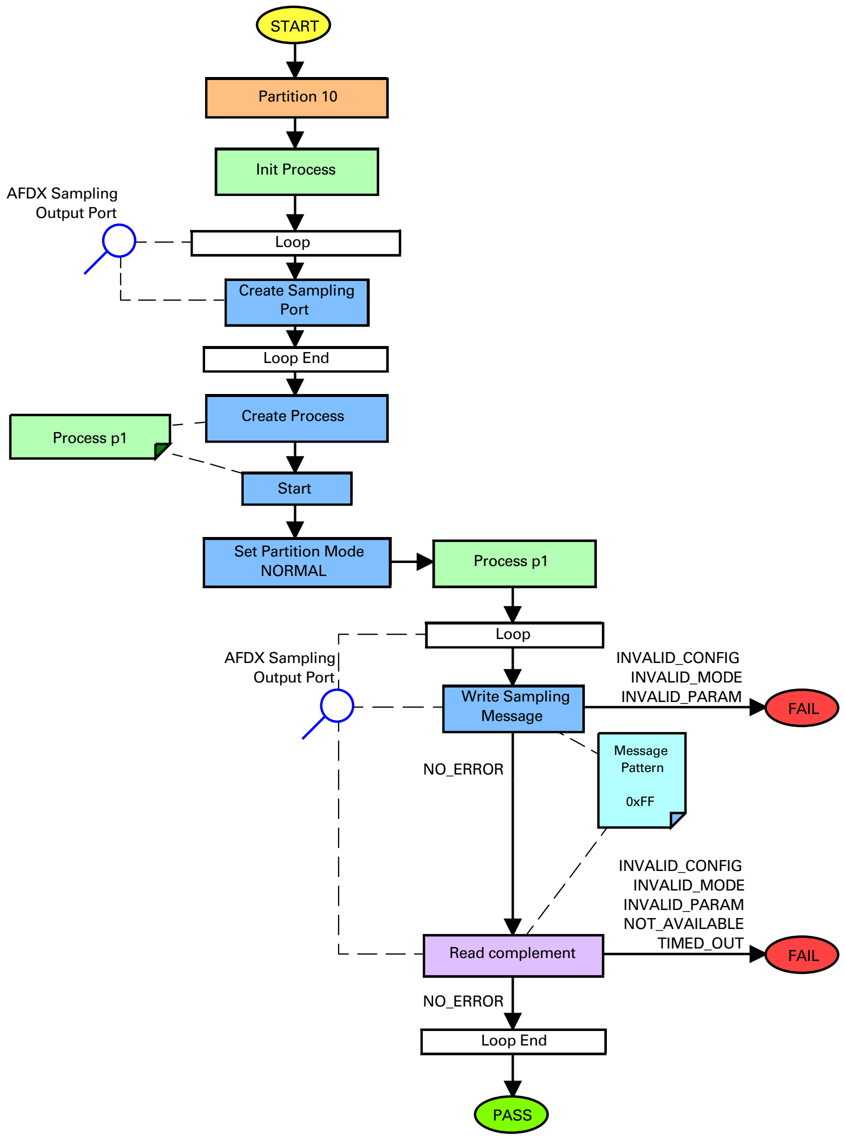


Figure A.3: AFDX port API test model

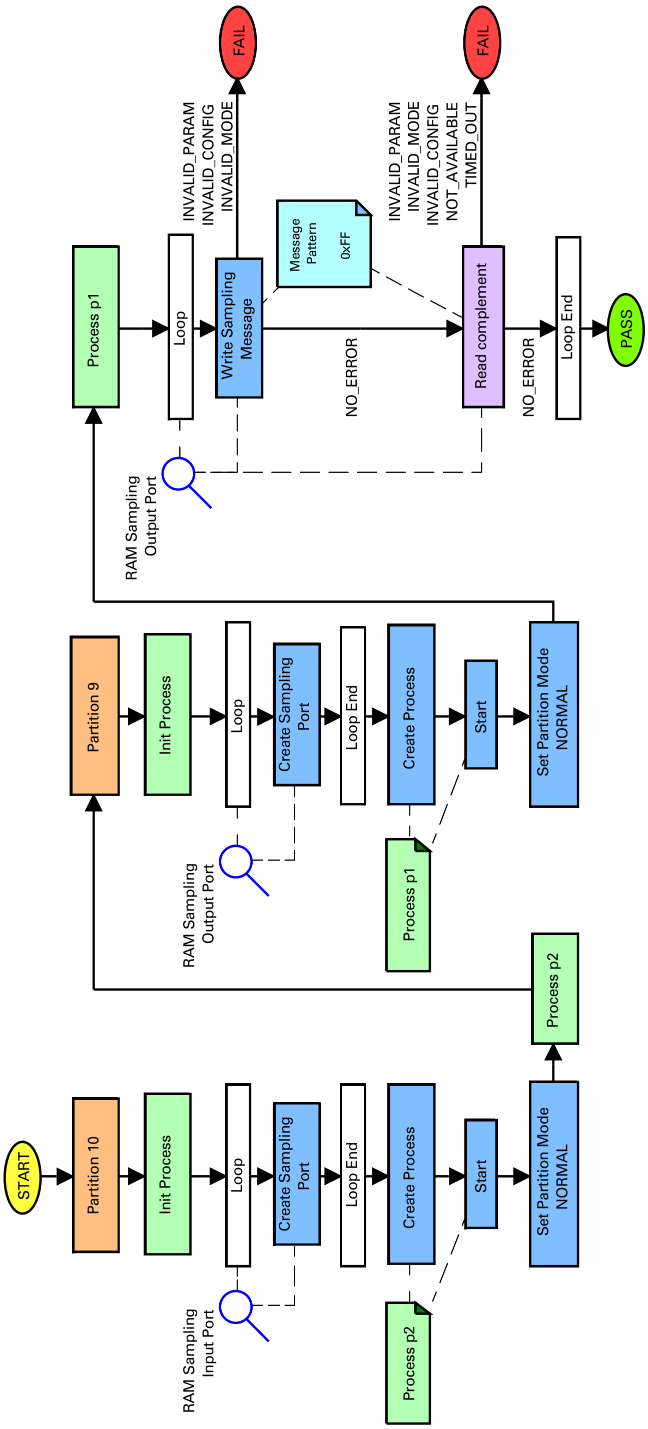


Figure A.4: RAM port API test model

A.1.1 Intermediate XML Representation

```xml
<?xml version=”1.0”?>
<behaviour xmlns=”http://www.scarlettproject.eu/tzi/behaviour” name=”Blackboard”>
  <resources>
    <process id=”3_1668” name=”p1” stacksize=”32768” priority=”1” period=”1000” ↩
         ↪ timecapacity=”500” deadline=”SOFT”/>
    <process id=”3_1767” name=”p2” stacksize=”32768” priority=”0” period=”1000” ↩
         ↪ timecapacity=”500” deadline=”SOFT”/>
    <blackboard id=”3_1610” name=”bb01” maxmsgsize=”64”/>
    <message id=”3_1840” pattern=”0xFF” size=”32”/>
  </resources>
  <nodes>
    <test_start id=”3_1527”/>
    <test_fail id=”3_1909”/>
    <test_pass id=”3_1912”/>
    <partition id=”3_1537” name=”9”/>
    <process id=”3_2105” name=”INIT”/>
    <process id=”3_2112” name=”p1”/>
    <process id=”3_2119” name=”p2”/>
    <Create_Blackboard id=”3_1596”/>
    <Create_Process id=”3_1617”/>
    <Create_Process id=”3_1782”/>
    <Display_Blackboard id=”3_1744”/>
    <Start id=”3_1692”/>
    <Start id=”3_1785”/>
    <Read_Blackboard id=”3_1865” timeout=”0”/>
    <Set_Partition_Mode id=”3_1715” mode=”NORMAL”/>
  </nodes>
  <parameters>
    <assign op=”3_1596” param=”3_1610”/>
    <assign op=”3_1865” param=”3_1610”/>
    <assign op=”3_1744” param=”3_1610”/>
    <assign op=”3_1744” param=”3_1840”/>
    <assign op=”3_1865” param=”3_1840”/>
    <assign op=”3_1617” param=”3_1668”/>
    <assign op=”3_1692” param=”3_1668”/>
    <assign op=”3_1782” param=”3_1767”/>
    <assign op=”3_1785” param=”3_1767”/>
  </parameters>
  <controlflow>
    <edge from=”3_1527” to=”3_1537”/>
    <edge from=”3_1596” to=”3_1617”/>
    <edge from=”3_1617” to=”3_1692”/>
    <edge from=”3_1692” to=”3_1782”/>
    <edge from=”3_1782” to=”3_1785”/>
    <edge from=”3_1785” to=”3_1715”/>
    <edge from=”3_1537” to=”3_2105”/>
    <edge from=”3_2105” to=”3_1596”/>
    <edge from=”3_1715” to=”3_2112”/>
    <edge from=”3_2112” to=”3_1744”/>
    <edge from=”3_1744” to=”3_2119”/>
    <edge from=”3_2119” to=”3_1865”/>
    <edge from=”3_1865” to=”3_1909”>
      <condition>INVALID_MODE</condition>
      <condition>INVALID_PARAM</condition>
      <condition>NOT_AVAILABLE</condition>
      <condition>TIMED_OUT</condition>
    </edge>
    <edge from=”3_1865” to=”3_1912”>
      <condition>NO_ERROR</condition>
    </edge>
  </controlflow>
</behaviour>
```

A.1.2 TTCN-3 Code

```
// Generated by ITML-B/C Test Generator 1.0 on 2014-Jun-21 16:26:06
// from model ”Blackboard.xml”
// Configuration:
//   ”COMMANDPORTS.csv”
//   ”L1G0000M09V001I001.csv”
//   ”L1G0000M10V001I001.csv”
//   ”L1L0000M09V001I001.csv”
//   ”L1L0000M10V001I001.csv”
//   ”L1M0000M00V001I001.csv”

module m_Blackboard {

  import from types all;
  import from TACP all;

  type component MTCType {
    port CmdPort cmd_p9;
  }

  type component SUTType {
    port TACP_Out_Port TZI_TA_P9_CMD_IN;
    port TACP_In_Port TZI_TA_P9_CMD_OUT;
  }

  function run_Blackboard() runs on MTCType {
    var charstring node;
    var RETURN_CODE_TYPE return_value;
    timer complement_timer;
    var integer p9_blackboard_bb01_id;
    template charstring p9_msg3_1840_pattern := ”0xFF0xFF0xFF0xFF0xFF0xFF0xFF0xFF”;
    var integer p9_process_p2_id;
    var integer p9_process_p1_id;

    // pre-fill data table
    cmd_p9.call(SET_DATA_TABLE_ENTRY:{0, ”bb01”, -}) {
      [] cmd_p9.getreply(SET_DATA_TABLE_ENTRY:{-, -, NO_ERROR});
    }
    cmd_p9.call(SET_DATA_TABLE_ENTRY:{1, p9_msg3_1840_pattern, -}) {
      [] cmd_p9.getreply(SET_DATA_TABLE_ENTRY:{-, -, NO_ERROR});
    }
    cmd_p9.call(SET_DATA_TABLE_ENTRY:{2, ”p2”, -}) {
      [] cmd_p9.getreply(SET_DATA_TABLE_ENTRY:{-, -, NO_ERROR});
    }
    cmd_p9.call(SET_DATA_TABLE_ENTRY:{3, ”p1”, -}) {
      [] cmd_p9.getreply(SET_DATA_TABLE_ENTRY:{-, -, NO_ERROR});
    }

    node := ”node_3_1527”;
    while (node != ””) {
      if (node == ”node_3_1527”) {
        // start
        node := ”node_3_1537”;
        continue;
      }

      if (node == ”node_3_1537”) {
        // partition 9
        node := ”node_3_2105”;
        continue;
      }

      if (node == ”node_3_2105”) {
        // process INIT
        cmd_p9.call(set_process_idx:{0}) {
```

```
      [] cmd_p9.getreply(set_process_idx:{-});
    }
    node := ”node_3_1596”;
    continue;
  }

  if (node == ”node_3_1596”) {
    // Create_Blackboard
    cmd_p9.call(CREATE_BLACKBOARD:{0, 64, -, -}) {
      [] cmd_p9.getreply(CREATE_BLACKBOARD:{-, -, ?, ?})
        -> param(-, -, p9_blackboard_bb01_id, return_value);
    }
    node := ”node_3_1617”;
    continue;
  }

  if (node == ”node_3_1617”) {
    // Create_Process
    cmd_p9.call(CREATE_PROCESS:{3, 0, 32768, 1, 1000, 500, SOFT, -, -}) {
      [] cmd_p9.getreply(CREATE_PROCESS:{-, -, -, -, -, -, -, ?, ?})
        -> param(-, -, -, -, -, -, -, p9_process_p1_id, return_value);
    }
    node := ”node_3_1692”;
    continue;
  }

  if (node == ”node_3_1692”) {
    // Start
    cmd_p9.call(START:{p9_process_p1_id, -}) {
      [] cmd_p9.getreply(START:{-, ?})
        -> param(-, return_value);
    }
    node := ”node_3_1782”;
    continue;
  }

  if (node == ”node_3_1782”) {
    // Create_Process
    cmd_p9.call(CREATE_PROCESS:{2, 0, 32768, 0, 1000, 500, SOFT, -, -}) {
      [] cmd_p9.getreply(CREATE_PROCESS:{-, -, -, -, -, -, -, ?, ?})
        -> param(-, -, -, -, -, -, -, p9_process_p2_id, return_value);
    }
    node := ”node_3_1785”;
    continue;
  }

  if (node == ”node_3_1785”) {
    // Start
    cmd_p9.call(START:{p9_process_p2_id, -}) {
      [] cmd_p9.getreply(START:{-, ?})
        -> param(-, return_value);
    }
    node := ”node_3_1715”;
    continue;
  }

  if (node == ”node_3_1715”) {
    // Set_Partition_Mode
    cmd_p9.call(SET_PARTITION_MODE:{NORMAL, -}) {
      [] cmd_p9.getreply(SET_PARTITION_MODE:{-, ?})
        -> param(-, return_value);
    }
    node := ”node_3_2112”;
    continue;
  }

  if (node == ”node_3_2112”) {
      // process p1
      cmd_p9.call(set_process_idx:{p9_process_p1_id}) {
        [] cmd_p9.getreply(set_process_idx:{-});
      }
      node := ”node_3_1744”;
      continue;
    }

    if (node == ”node_3_1744”) {
      // Display_Blackboard
      cmd_p9.call(DISPLAY_BLACKBOARD:{p9_blackboard_bb01_id, 1, 32, -}) {
        [] cmd_p9.getreply(DISPLAY_BLACKBOARD:{-, -, -, ?})
          -> param(-, -, -, return_value);
      }
      node := ”node_3_2119”;
      continue;
    }

    if (node == ”node_3_2119”) {
      // process p2
      cmd_p9.call(set_process_idx:{p9_process_p2_id}) {
        [] cmd_p9.getreply(set_process_idx:{-});
      }
      node := ”node_3_1865”;
      continue;
    }

    if (node == ”node_3_1865”) {
      // Read_Blackboard
      cmd_p9.call(READ_BLACKBOARD:{p9_blackboard_bb01_id, 0, 1, -, -}) {
        [] cmd_p9.getreply(READ_BLACKBOARD:{-, -, -, ?, ?})
          -> param(-, -, -, -, return_value);
      }
      if (return_value == INVALID_MODE or
        return_value == INVALID_PARAM or
        return_value == NOT_AVAILABLE or
        return_value == TIMED_OUT) {
        node := ”node_3_1909”;
        continue;
      }
      if (return_value == NO_ERROR) {
        node := ”node_3_1912”;
        continue;
      }
      node := ””;
      continue;
    }

    if (node == ”node_3_1909”) {
      setverdict(fail);
      node := ””;
      continue;
    }

    if (node == ”node_3_1912”) {
      setverdict(pass);
      node := ””;
      continue;
    }

  }
}

testcase Blackboard() runs on MTCType system SUTType {
  timer tacp_sync := 1.0;
  var TACPHandler p9_handler;
```

```
    p9_handler := TACPHandler.create;

    map(p9_handler:tacpToCPM, system:TZI_TA_P9_CMD_IN);
    map(p9_handler:tacpFromCPM, system:TZI_TA_P9_CMD_OUT);
    connect(mtc:cmd_p9, p9_handler:cmd);


    p9_handler.start(runHandler());
    tacp_sync.start;
    tacp_sync.timeout;
    run_Blackboard();
  }
}
```

Due to their size the intermediate XML representations and the generated TTCN-3 test procedures for the other models are not reproduced here.

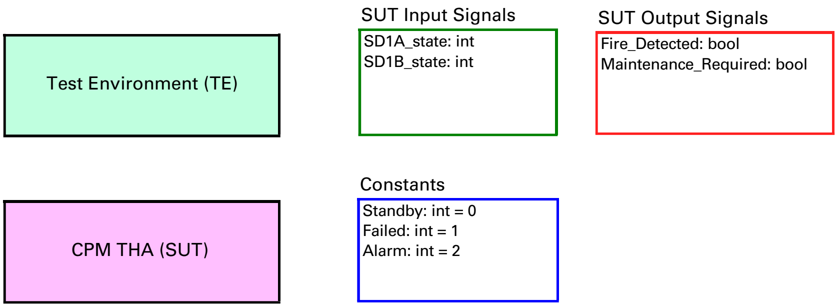


Figure A.5: System diagram

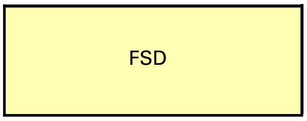


Figure A.6: System Under Test component diagram

A.2 ITML-A

The remainder of this appendix shows a complete ITML-A example: a Fire and Smoke Detection application (FSD), as commonly used in modern commercial aircraft. The system consists of smoke detectors (SD) which are installed in different areas of the aircraft, e. g. lavatories, galleys, or cargo compartments. The smoke detectors are connected to a data bus, which links them to the module hosting the FSD application. The application receives status updates from the smoke detectors and determines if a fire has been detected. Upon detection the application signals the flight crew and triggers fire extinguishing measures. If a smoke detector fails, the application gives a signal that on-ground maintenance is required.

Figure A.5 shows the system diagram for the FSD application. It consists of an SUT component representing a CPM and the test environment (TE). The signal lists show the input and output signals that the SUT provides towards the TE. In particular, `SD1A_state` and `SD1B_state` are smoke detector status inputs, while `Fire_Detected` and `Maintenance_Required` are outputs from the application. The constants list contains values for the encoding of the smoke detector status in the SD status signals.

Figure A.6 shows the interior of the SUT component: The SUT decomposes into a single component representing the FSD application running on the module. Analogously, figure A.7 shows the decomposition of the TE: It consists of a simulation component representing smoke detector SD1.

The SD1 component decomposes further into another component dia-

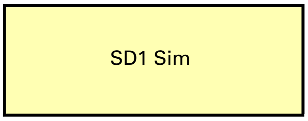


Figure A.7: Test Environment component diagram

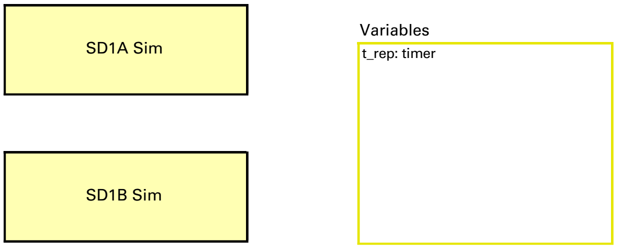


Figure A.8: SD1 simulation component diagram

gram shown in figure A.8. The smoke detector SD1 consists of two individual detectors, SD1A and SD1B. Therefore, SD1 represents a so-called dual smoke detector, providing fault tolerance. Inside this component a timer, `t_rep`, is defined. Its purpose will be explained below.

The simulation components SD1A and SD1B decompose into the statecharts shown in figure A.9 and figure A.10, respectively. Each detector has three primary states, *Standby*, *Alarm*, and *Failed*. The entry action of each state sets the SD state SUT input signal to the respective value. *Standby* is the initial state of each SD. Normally, *Failed* would be a terminal location (i. e. the detector has to be replaced). In order to not have a dead end in this model, a *Repair* location has been added. This simulates the replacement of the smoke detector. Note that the guard condition only allows the transition to the repair state when both detectors of the dual SD have failed. The entry action starts the repair timer `t_rep` mentioned earlier. As soon as it expires, both detectors can transition back to *Standby*.

The remaining figure A.11 shows the statechart to which the FSD component inside the SUT decomposes. It consists of five locations. The three locations in the centre row correspond to both detectors having the same state, i. e. being in *Alarm*, *Standby*, and *Failed* states, respectively. The location on the top, *SD Inconsistent*, is entered if one SD is in *Alarm* state while the other is in *Standby*. This means that the application has detected an inconsistency in the SD states. Upon entry, the application starts the timer `t`. If it expires, the whole detector is considered failed due to the inconsistency. If, however, the second detector also changes from *Standby* to *Alarm* state before the timer expires, the application enters the *Alarm* location, and upon entry sets its `Fire_Detected` output to true. Conversely, the application

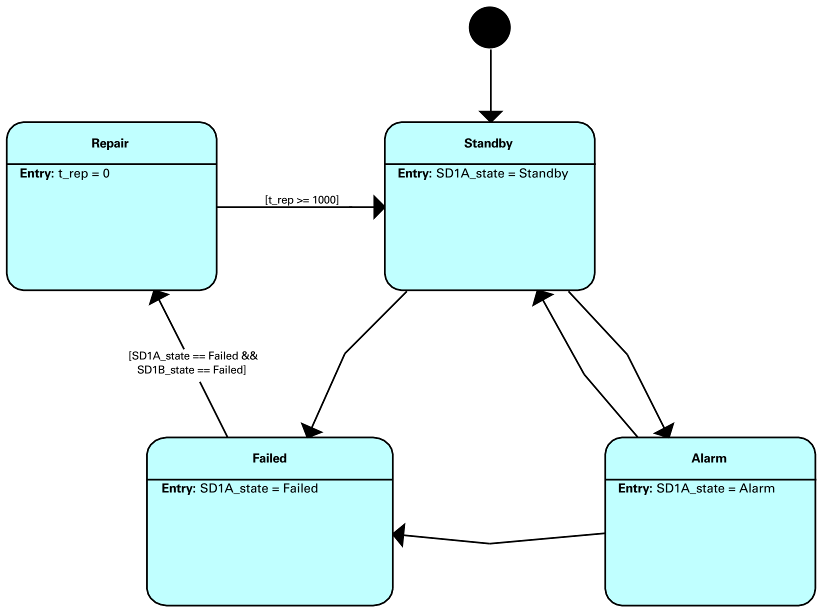


Figure A.9: SD1A simulation statechart

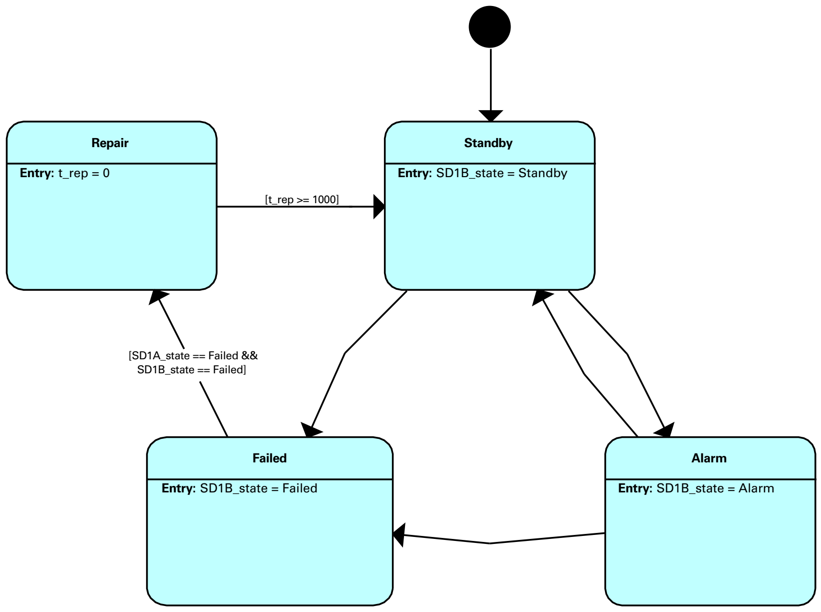


Figure A.10: SD1B simulation statechart

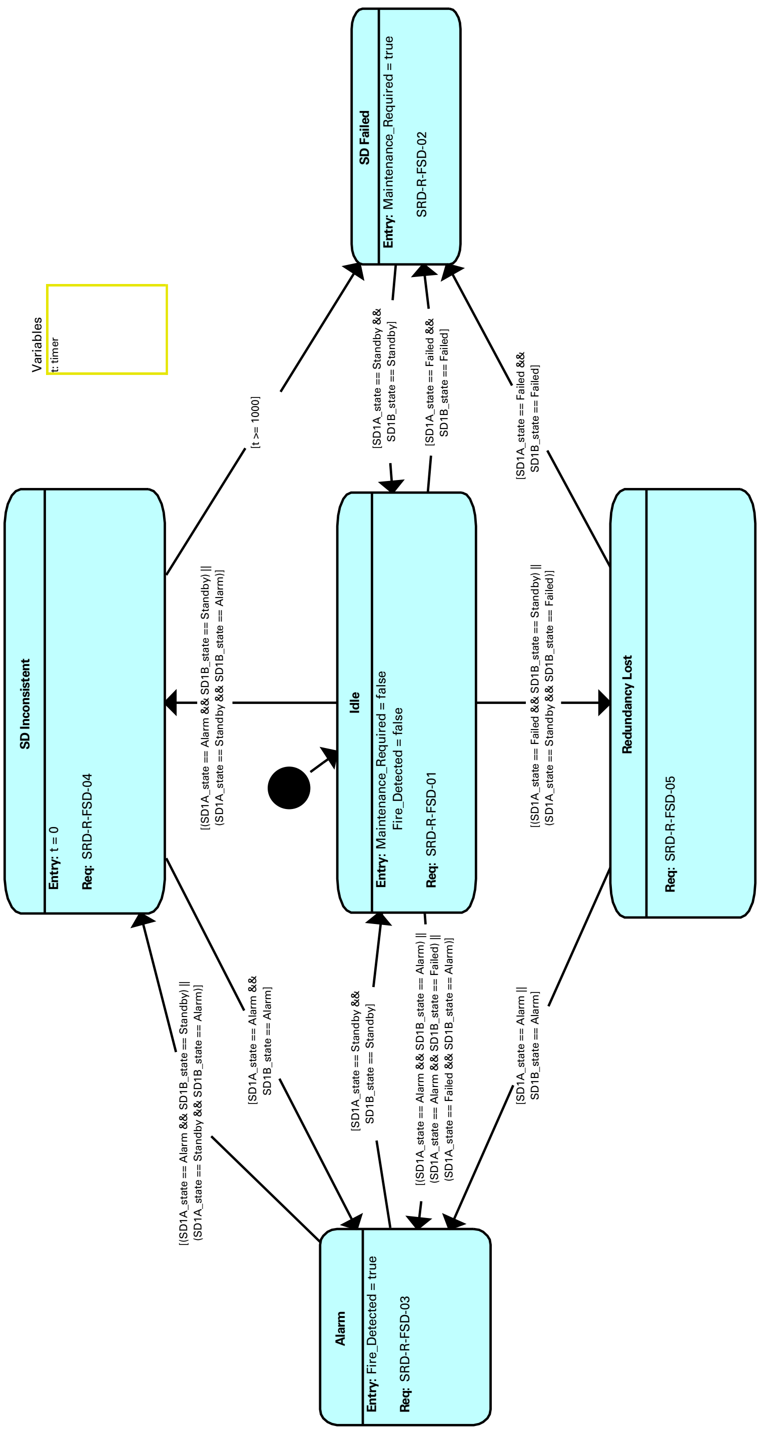

SD Failed
Entry: Maintenance_Required = true
SRD-R-FSD-02
Variables
t: timer
SD Inconsistent
Entry: t = 0
Req: SRD-R-FSD-04
Idle
Entry: Maintenance_Required = false
Fire_Detected = false
Req: SRD-R-FSD-01
Redundancy Lost
Req: SRD-R-FSD-05
Alarm
Entry: Fire_Detected = true
Req: SRD-R-FSD-03
[t >= 1000]
[SD1A_state == Standby && SD1B_state == Standby]
[SD1A_state == Failed && SD1B_state == Failed]
[SD1A_state == Failed && SD1B_state == Failed]
[(SD1A_state == Alarm && SD1B_state == Standby) || (SD1A_state == Standby && SD1B_state == Alarm)]
[(SD1A_state == Failed && SD1B_state == Standby) || (SD1A_state == Standby && SD1B_state == Failed)]
[(SD1A_state == Alarm && SD1B_state == Standby) || (SD1A_state == Standby && SD1B_state == Alarm)]
[SD1A_state == Alarm && SD1B_state == Alarm]
[SD1A_state == Standby && SD1B_state == Standby]
[(SD1A_state == Alarm && SD1B_state == Alarm) || (SD1A_state == Alarm && SD1B_state == Failed) || (SD1A_state == Failed && SD1B_state == Alarm)]
[SD1A_state == Alarm || SD1B_state == Alarm]

Figure A.11: FSD application statechart

enters the location on the bottom, *Redundancy Lost*, if one detector goes to *Failed* state while the other is in *Standby*. In this state, the single *Alarm* signal of the other (good) detector is sufficient to transition to the *Alarm location* and report the detection of fire. If, however, the good detector also goes into *Failed* state, the application transitions into the *SD Failed* location on the right and sets the `Maintenance_Required` output signal to true.

Appendix A.2.1 shows the intermediate XML representation that was created from the system diagram (figure A.5) by the generator shown in appendix E.5. The root element represents the system. It contains the `sut` and the `testenv` elements. Each of these can have nested `component` elements. Each nesting level has its respective `signal`, `constant`, and `variable` lists. The innermost components (`FSD`, `SD1A Sim`, and `SD1B Sim`) contain `statechart` elements. Each one of these contains two sections: `locations` and `transitions`. The former contains a `location` element for each location of the statechart, the latter contains a `transition` element for each transition between locations, including the respective guard condition.

Appendix A.2.2 finally shows the TTCN-3 test procedure generated by the ITML-A test generator from the intermediate XML representation in appendix A.2.1. It contains the appropriate test oracle function for the FSD application as well as the stimulator function which provides the SUT inputs required to reach full location and transition coverage of the SUT model.

A.2.1 Intermediate XML Representation

```
<?xml version="1.0"?>
<system xmlns="http://www.scarlettproject.eu/tzi/system" name="I/O Intensive ↩
     ↪ Demonstrator">
  <sut_inputs>
    <signal name="SD1A_state" type="int" minimum="0" maximum="2" default="0"/>
    <signal name="SD1B_state" type="int" minimum="0" maximum="2" default="0"/>
  </sut_inputs>
  <sut_outputs>
    <signal name="Fire_Detected" type="bool" minimum="0" maximum="1" default="0"/>
    <signal name="Maintenance_Required" type="bool" minimum="0" maximum="1" ↩
         ↪ default="0"/>
  </sut_outputs>
  <constants>
    <constant name="Standby" type="int" value="0"/>
    <constant name="Failed" type="int" value="1"/>
    <constant name="Alarm" type="int" value="2"/>
  </constants>
  <sut name="CPM THA">
    <variables>
    </variables>
    <constants>
    </constants>
    <component name="FSD">
      <variables>
        <variable name="t" type="timer" minimum="0" maximum="0" default="0"/>
      </variables>
      <constants>
      </constants>
      <statechart>
        <locations>
          <start_location id="4_1277"/>
          <location id="4_1179" name="Alarm">
            <req>SRD-R-FSD-03</req>
            <entry>Fire_Detected = true; </entry>
          </location>
          <location id="4_1194" name="Idle">
            <req>SRD-R-FSD-01</req>
            <entry>Maintenance_Required = false; Fire_Detected = false; </entry>
          </location>
          <location id="4_1230" name="Redundancy Lost">
            <req>SRD-R-FSD-05</req>
          </location>
          <location id="4_1245" name="SD Failed">
            <req>SRD-R-FSD-02</req>
            <entry>Maintenance_Required = true; </entry>
          </location>
          <location id="4_1213" name="SD Inconsistent">
            <req>SRD-R-FSD-04</req>
            <entry>t = 0; </entry>
          </location>
        </locations>
        <transitions>
          <transition id="4_1295" name="" from="4_1277" to="4_1194">
          </transition>
          <transition id="4_1310" name="" from="4_1194" to="4_1179">
            <cond>(SD1A_state == Alarm &amp;&amp; SD1B_state == Alarm) || ↩
                 ↪ (SD1A_state == Alarm &amp;&amp; SD1B_state == Failed) || ↩
                 ↪ (SD1A_state == Failed &amp;&amp; SD1B_state == Alarm)</cond>
          </transition>
          <transition id="4_1325" name="" from="4_1213" to="4_1179">
            <cond>SD1A_state == Alarm &amp;&amp; SD1B_state == Alarm</cond>
          </transition>
          <transition id="4_1340" name="" from="4_1194" to="4_1213">
            <cond>(SD1A_state == Alarm &amp;&amp; SD1B_state == Standby) || ↩
```

```
               ↪ (SD1A_state == Standby &amp;&amp; SD1B_state == Alarm)</cond>
        </transition>
        <transition id="4_1355" name="" from="4_1230" to="4_1179">
          <cond>SD1A_state == Alarm || SD1B_state == Alarm</cond>
        </transition>
        <transition id="4_1370" name="" from="4_1194" to="4_1230">
          <cond>(SD1A_state == Failed &amp;&amp; SD1B_state == Standby) || ↩
               ↪ (SD1A_state == Standby &amp;&amp; SD1B_state == Failed)</cond>
        </transition>
        <transition id="4_1385" name="" from="4_1194" to="4_1245">
          <cond>SD1A_state == Failed &amp;&amp; SD1B_state == Failed</cond>
        </transition>
        <transition id="4_1400" name="" from="4_1213" to="4_1245">
          <cond>t &gt;= 1000</cond>
        </transition>
        <transition id="4_1415" name="" from="4_1230" to="4_1245">
          <cond>SD1A_state == Failed &amp;&amp; SD1B_state == Failed</cond>
        </transition>
        <transition id="4_2118" name="" from="4_1179" to="4_1213">
          <cond>(SD1A_state == Alarm &amp;&amp; SD1B_state == Standby) || ↩
               ↪ (SD1A_state == Standby &amp;&amp; SD1B_state == Alarm)</cond>
        </transition>
        <transition id="4_2135" name="" from="4_1179" to="4_1194">
          <cond>SD1A_state == Standby &amp;&amp; SD1B_state == Standby</cond>
        </transition>
        <transition id="4_2279" name="" from="4_1245" to="4_1194">
          <cond>SD1A_state == Standby &amp;&amp; SD1B_state == Standby</cond>
        </transition>
      </transitions>
    </statechart>
  </component>
</sut>
<testenv name="ADS2">
  <variables>
  </variables>
  <constants>
  </constants>
  <component name="SD1 Sim">
    <variables>
      <variable name="t_rep" type="timer" minimum="0" maximum="0" default="0"/>
    </variables>
    <constants>
    </constants>
    <component name="SD1A Sim">
      <variables>
      </variables>
      <constants>
      </constants>
      <statechart>
        <locations>
          <start_location id="4_708"/>
          <location id="4_756" name="Alarm">
            <entry>SD1A_state = Alarm; </entry>
          </location>
          <location id="4_726" name="Failed">
            <entry>SD1A_state = Failed; </entry>
          </location>
          <location id="4_2169" name="Repair">
            <entry>t_rep = 0; </entry>
          </location>
          <location id="4_695" name="Standby">
            <entry>SD1A_state = Standby; </entry>
          </location>
        </locations>
        <transitions>
          <transition id="4_711" name="" from="4_708" to="4_695">
          </transition>
```

```
            <transition id="4_769" name="" from="4_695" to="4_756">
            </transition>
            <transition id="4_784" name="" from="4_695" to="4_726">
            </transition>
            <transition id="4_799" name="" from="4_756" to="4_695">
            </transition>
            <transition id="4_829" name="" from="4_756" to="4_726">
            </transition>
            <transition id="4_2184" name="" from="4_726" to="4_2169">
              <cond>SD1A_state == Failed &amp;&amp; SD1B_state == Failed</cond>
            </transition>
            <transition id="4_2201" name="" from="4_2169" to="4_695">
              <cond>t_rep &gt;= 1000</cond>
            </transition>
          </transitions>
        </statechart>
      </component>
      <component name="SD1B Sim">
        <variables>
        </variables>
        <constants>
        </constants>
        <statechart>
          <locations>
            <start_location id="4_1431"/>
            <location id="4_1441" name="Alarm">
              <entry>SD1B_state = Alarm; </entry>
            </location>
            <location id="4_1467" name="Failed">
              <entry>SD1B_state = Failed; </entry>
            </location>
            <location id="4_2218" name="Repair">
              <entry>t_rep = 0; </entry>
            </location>
            <location id="4_1493" name="Standby">
              <entry>SD1B_state = Standby; </entry>
            </location>
          </locations>
          <transitions>
            <transition id="4_1521" name="" from="4_1441" to="4_1467">
            </transition>
            <transition id="4_1561" name="" from="4_1493" to="4_1441">
            </transition>
            <transition id="4_1581" name="" from="4_1441" to="4_1493">
            </transition>
            <transition id="4_1601" name="" from="4_1431" to="4_1493">
            </transition>
            <transition id="4_1621" name="" from="4_1493" to="4_1467">
            </transition>
            <transition id="4_2233" name="" from="4_1467" to="4_2218">
              <cond>SD1A_state == Failed &amp;&amp; SD1B_state == Failed</cond>
            </transition>
            <transition id="4_2250" name="" from="4_2218" to="4_1493">
              <cond>t_rep &gt;= 1000</cond>
            </transition>
          </transitions>
        </statechart>
      </component>
    </component>
  </testenv>
</system>
```

A.2.2 TTCN-3 Code

```
/*
 * Generated by ITML-A Test Generator 9.0-1.2.0-x64.83c0017.dev on 2014-Jul-05 14:14:09
 * from model IO Intensive Demonstrator.xml
 */

module m_main {

  type integer clock;

  type port Fire_DetectedPort message {
    in boolean
  }

  type port Maintenance_RequiredPort message {
    in boolean
  }

  type port SD1A_statePort message {
    out integer
  }

  type port SD1B_statePort message {
    out integer
  }

  type component MTCType {
    // ports
    port Fire_DetectedPort Fire_Detected;
    port Maintenance_RequiredPort Maintenance_Required;
    port SD1A_statePort SD1A_state;
    port SD1B_statePort SD1B_state;

    // I/O variables
    var boolean IOpre_Fire_Detected := false;
    var boolean IOpost_Fire_Detected := false;
    var boolean IOpre_Maintenance_Required := false;
    var boolean IOpost_Maintenance_Required := false;
    var integer IOpre_SD1A_state := 0;
    var integer IOpost_SD1A_state := 0;
    var integer IOpre_SD1B_state := 0;
    var integer IOpost_SD1B_state := 0;

    // state variables
    var clock Statepre_sut_FSD_t := 0;
    var clock Statepost_sut_FSD_t := 0;
    var clock Statepre_te__SD1_Sim_t_rep := 0;
    var clock Statepost_te__SD1_Sim_t_rep := 0;

    // oracle variables
    var integer Fire_Detected_counter := 0;
    var boolean Fire_Detected_passed := false;
    var boolean Fire_Detected_failed := false;
    var boolean Fire_Detected_expected := false;
    var boolean Fire_Detected_expectedOld := false;
    var integer Maintenance_Required_counter := 0;
    var boolean Maintenance_Required_passed := false;
    var boolean Maintenance_Required_failed := false;
    var boolean Maintenance_Required_expected := false;
    var boolean Maintenance_Required_expectedOld := false;
    var boolean haveDiscreteTrans := false;
    const float cycle_period := 0.1;
    var clock _timeTick := 0;
    timer cycleTimer := cycle_period;

    // oracle location
    var integer sut_FSD_currentLocation := 7;

    // stimulator location
    var integer stimulator_location := 0;
  }

  // operations

  // oracles
  function _ora_FSD() runs on MTCType
  {
    var integer cycleCtr := 0;
    var boolean triggered := false;
```

```
while (not triggered)
{
  /* Handle location 'FSD.Start' */
  if ((sut_FSD_currentLocation == 7) and (not triggered))
  {
    if (not triggered)
    {
      haveDiscreteTrans := true;
      Maintenance_Required_expected := false;
      Fire_Detected_expected := false;
      /* New location is 'FSD.Idle' */
      sut_FSD_currentLocation := 10;
      /**
       * Model coverage goal: basic control state coverage
       *
       * @tag       TC-mbt-fsd-BCS-0002 Cover basic control state Idle
       * @condition TRUE
       * @event     Component IMR.SystemUnderTest.FSD
       *            reaches basic control state IMR.SystemUnderTest.FSD.FSD.Idle
       * @expected  The actions associated with the transition entering
       *            this control state, and the control state's entry actions are
       *            performed as specified in the model.
       *
       * @note These checks are performed by the test oracles associated
       *       with component IMR.SystemUnderTest.FSD
       * @req SRD-R-FSD-01
       */
      log("TC-mbt-fsd-BCS-0002");
      triggered := true;
    }
    triggered := true;
  }
  /* Handle location 'FSD.Alarm' */
  if ((sut_FSD_currentLocation == 8) and (not triggered))
  {
    if ((not triggered) and (((IOpre_SD1A_state == 2) and (IOpre_SD1B_state == 0)) or ↩
          ↪ ((IOpre_SD1A_state == 0) and (IOpre_SD1B_state == 2))))
    {
      haveDiscreteTrans := true;
      Statepost_sut_FSD_t := (_timeTick + 0);
      /* New location is 'FSD.SD_Inconsistent' */
      sut_FSD_currentLocation := 16;
      /**
       * Model coverage goal : transition coverage
       * Cover transition of component IMR.SystemUnderTest.FSD
       *   FSD.Alarm
       *       -- [ (((IMR.SD1A_state == 2) && (IMR.SD1B_state == 0)) || ((IMR.SD1A_state ↩
            ↪ == 0) && (IMR.SD1B_state == 2))) ] -->
       *   FSD.SD_Inconsistent
       *
       * @tag       TC-mbt-fsd-TR-0001 Cover transition Alarm --> SD_Inconsistent
       * @condition Component IMR.SystemUnderTest.FSD
       *            resides in control state IMR.SystemUnderTest.FSD.FSD.Alarm
       * @event     Trigger condition for specified transition becomes true
       * @expected  The actions associated with the transition specified above,
       *            and the target state's entry actions are
       *            performed as specified in the model.
       *
       * @note These checks are performed by the test oracles associated
       *       with component IMR.SystemUnderTest.FSD
       * @req SRD-R-FSD-03
       */
      /**
       * Model coverage goal: basic control state coverage
       *
       * @tag       TC-mbt-fsd-BCS-0005 Cover basic control state SD_Inconsistent
       * @condition TRUE
       * @event     Component IMR.SystemUnderTest.FSD
       *            reaches basic control state IMR.SystemUnderTest.FSD.FSD.SD_Inconsistent
       * @expected  The actions associated with the transition entering
       *            this control state, and the control state's entry actions are
       *            performed as specified in the model.
       *
       * @note These checks are performed by the test oracles associated
       *       with component IMR.SystemUnderTest.FSD
       * @req SRD-R-FSD-04
      */
     log(”TC-mbt-fsd-TR-0001, TC-mbt-fsd-BCS-0005”);
     triggered := true;
   }
   if ((not triggered) and ((IOpre_SD1A_state == 0) and (IOpre_SD1B_state == 0)))
   {
     haveDiscreteTrans := true;
     Maintenance_Required_expected := false;
     Fire_Detected_expected := false;
     /* New location is ’FSD.Idle’ */
     sut_FSD_currentLocation := 10;
     /**
      * Model coverage goal : transition coverage
      * Cover transition of component IMR.SystemUnderTest.FSD
      *   FSD.Alarm
      *       -- [ ((IMR.SD1A_state == 0) && (IMR.SD1B_state == 0)) ] -->
      *   FSD.Idle
      *
      * @tag       TC-mbt-fsd-TR-0002 Cover transition Alarm --> Idle
      * @condition Component IMR.SystemUnderTest.FSD
      *            resides in control state IMR.SystemUnderTest.FSD.FSD.Alarm
      * @event     Trigger condition for specified transition becomes true
      * @expected  The actions associated with the transition specified above,
      *            and the target state’s entry actions are
      *            performed as specified in the model.
      *
      * @note These checks are performed by the test oracles associated
      *       with component IMR.SystemUnderTest.FSD
      * @req SRD-R-FSD-03
      */
     /**
      * Model coverage goal: basic control state coverage
      *
      * @tag       TC-mbt-fsd-BCS-0002 Cover basic control state Idle
      * @condition TRUE
      * @event     Component IMR.SystemUnderTest.FSD
      *            reaches basic control state IMR.SystemUnderTest.FSD.FSD.Idle
      * @expected  The actions associated with the transition entering
      *            this control state, and the control state’s entry actions are
      *            performed as specified in the model.
      *
      * @note These checks are performed by the test oracles associated
      *       with component IMR.SystemUnderTest.FSD
      * @req SRD-R-FSD-01
      */
     log(”TC-mbt-fsd-TR-0002, TC-mbt-fsd-BCS-0002”);
     triggered := true;
   }
   triggered := true;
 }
 /* Handle location ’FSD.Idle’ */
 if ((sut_FSD_currentLocation == 10) and (not triggered))
 {
   if ((not triggered) and ((((IOpre_SD1A_state == 2) and (IOpre_SD1B_state == 2)) or ↩
         ↪ ((IOpre_SD1A_state == 2) and (IOpre_SD1B_state == 1))) or ((IOpre_SD1A_state ↩
         ↪ == 1) and (IOpre_SD1B_state == 2))))
   {
     haveDiscreteTrans := true;
     Fire_Detected_expected := true;
     /* New location is ’FSD.Alarm’ */
     sut_FSD_currentLocation := 8;
     /**
      * Model coverage goal : transition coverage
      * Cover transition of component IMR.SystemUnderTest.FSD
      *   FSD.Idle
      *       -- [ ((((IMR.SD1A_state == 2) && (IMR.SD1B_state == 2)) || ((IMR.SD1A_state ↩
            ↪ == 2) && (IMR.SD1B_state == 1))) || ((IMR.SD1A_state == 1) && ↩
            ↪ (IMR.SD1B_state == 2))) ] -->
      *   FSD.Alarm
      *
      * @tag       TC-mbt-fsd-TR-0003 Cover transition Idle --> Alarm
      * @condition Component IMR.SystemUnderTest.FSD
      *            resides in control state IMR.SystemUnderTest.FSD.FSD.Idle
      * @event     Trigger condition for specified transition becomes true
      * @expected  The actions associated with the transition specified above,
      *            and the target state’s entry actions are
     *            performed as specified in the model.
     *
     * @note These checks are performed by the test oracles associated
     *       with component IMR.SystemUnderTest.FSD
     * @req SRD-R-FSD-01
     */
    /**
     * Model coverage goal: basic control state coverage
     *
     * @tag       TC-mbt-fsd-BCS-0001 Cover basic control state Alarm
     * @condition TRUE
     * @event     Component IMR.SystemUnderTest.FSD
     *            reaches basic control state IMR.SystemUnderTest.FSD.FSD.Alarm
     * @expected  The actions associated with the transition entering
     *            this control state, and the control state's entry actions are
     *            performed as specified in the model.
     *
     * @note These checks are performed by the test oracles associated
     *       with component IMR.SystemUnderTest.FSD
     * @req SRD-R-FSD-03
     */
    log("TC-mbt-fsd-TR-0003, TC-mbt-fsd-BCS-0001");
    triggered := true;
  }
  if ((not triggered) and (((IOpre_SD1A_state == 2) and (IOpre_SD1B_state == 0)) or ↩
      ↪ ((IOpre_SD1A_state == 0) and (IOpre_SD1B_state == 2))))
  {
    haveDiscreteTrans := true;
    Statepost_sut_FSD_t := (_timeTick + 0);
    /* New location is 'FSD.SD_Inconsistent' */
    sut_FSD_currentLocation := 16;
    /**
     * Model coverage goal : transition coverage
     * Cover transition of component IMR.SystemUnderTest.FSD
     *   FSD.Idle
     *       -- [ (((IMR.SD1A_state == 2) && (IMR.SD1B_state == 0)) || ((IMR.SD1A_state ↩
          ↪ == 0) && (IMR.SD1B_state == 2))) ] -->
     *   FSD.SD_Inconsistent
     *
     * @tag       TC-mbt-fsd-TR-0004 Cover transition Idle --> SD_Inconsistent
     * @condition Component IMR.SystemUnderTest.FSD
     *            resides in control state IMR.SystemUnderTest.FSD.FSD.Idle
     * @event     Trigger condition for specified transition becomes true
     * @expected  The actions associated with the transition specified above,
     *            and the target state's entry actions are
     *            performed as specified in the model.
     *
     * @note These checks are performed by the test oracles associated
     *       with component IMR.SystemUnderTest.FSD
     * @req SRD-R-FSD-01
     */
    /**
     * Model coverage goal: basic control state coverage
     *
     * @tag       TC-mbt-fsd-BCS-0005 Cover basic control state SD_Inconsistent
     * @condition TRUE
     * @event     Component IMR.SystemUnderTest.FSD
     *            reaches basic control state IMR.SystemUnderTest.FSD.FSD.SD_Inconsistent
     * @expected  The actions associated with the transition entering
     *            this control state, and the control state's entry actions are
     *            performed as specified in the model.
     *
     * @note These checks are performed by the test oracles associated
     *       with component IMR.SystemUnderTest.FSD
     * @req SRD-R-FSD-04
     */
    log("TC-mbt-fsd-TR-0004, TC-mbt-fsd-BCS-0005");
    triggered := true;
  }
  if ((not triggered) and (((IOpre_SD1A_state == 1) and (IOpre_SD1B_state == 0)) or ↩
      ↪ ((IOpre_SD1A_state == 0) and (IOpre_SD1B_state == 1))))
  {
    haveDiscreteTrans := true;
    /* New location is 'FSD.Redundancy_Lost' */
    sut_FSD_currentLocation := 13;
    /**
```

```
   * Model coverage goal : transition coverage
   * Cover transition of component IMR.SystemUnderTest.FSD
   *   FSD.Idle
   *       -- [ (((IMR.SD1A_state == 1) && (IMR.SD1B_state == 0)) || ((IMR.SD1A_state ↩
        ↪ == 0) && (IMR.SD1B_state == 1))) ] -->
   *   FSD.Redundancy_Lost
   *
   * @tag       TC-mbt-fsd-TR-0005 Cover transition Idle --> Redundancy_Lost
   * @condition Component IMR.SystemUnderTest.FSD
   *            resides in control state IMR.SystemUnderTest.FSD.FSD.Idle
   * @event     Trigger condition for specified transition becomes true
   * @expected  The actions associated with the transition specified above,
   *            and the target state’s entry actions are
   *            performed as specified in the model.
   *
   * @note These checks are performed by the test oracles associated
   *       with component IMR.SystemUnderTest.FSD
   * @req SRD-R-FSD-01
   */
  /**
   * Model coverage goal: basic control state coverage
   *
   * @tag       TC-mbt-fsd-BCS-0003 Cover basic control state Redundancy_Lost
   * @condition TRUE
   * @event     Component IMR.SystemUnderTest.FSD
   *            reaches basic control state IMR.SystemUnderTest.FSD.FSD.Redundancy_Lost
   * @expected  The actions associated with the transition entering
   *            this control state, and the control state’s entry actions are
   *            performed as specified in the model.
   *
   * @note These checks are performed by the test oracles associated
   *       with component IMR.SystemUnderTest.FSD
   * @req SRD-R-FSD-05
   */
  log(”TC-mbt-fsd-TR-0005, TC-mbt-fsd-BCS-0003”);
  triggered := true;
}
if ((not triggered) and ((IOpre_SD1A_state == 1) and (IOpre_SD1B_state == 1)))
{
  haveDiscreteTrans := true;
  Maintenance_Required_expected := true;
  /* New location is ’FSD.SD_Failed’ */
  sut_FSD_currentLocation := 14;
  /**
   * Model coverage goal : transition coverage
   * Cover transition of component IMR.SystemUnderTest.FSD
   *   FSD.Idle
   *       -- [ ((IMR.SD1A_state == 1) && (IMR.SD1B_state == 1)) ] -->
   *   FSD.SD_Failed
   *
   * @tag       TC-mbt-fsd-TR-0006 Cover transition Idle --> SD_Failed
   * @condition Component IMR.SystemUnderTest.FSD
   *            resides in control state IMR.SystemUnderTest.FSD.FSD.Idle
   * @event     Trigger condition for specified transition becomes true
   * @expected  The actions associated with the transition specified above,
   *            and the target state’s entry actions are
   *            performed as specified in the model.
   *
   * @note These checks are performed by the test oracles associated
   *       with component IMR.SystemUnderTest.FSD
   * @req SRD-R-FSD-01
   */
  /**
   * Model coverage goal: basic control state coverage
   *
   * @tag       TC-mbt-fsd-BCS-0004 Cover basic control state SD_Failed
   * @condition TRUE
   * @event     Component IMR.SystemUnderTest.FSD
   *            reaches basic control state IMR.SystemUnderTest.FSD.FSD.SD_Failed
   * @expected  The actions associated with the transition entering
   *            this control state, and the control state’s entry actions are
   *            performed as specified in the model.
   *
   * @note These checks are performed by the test oracles associated
   *       with component IMR.SystemUnderTest.FSD
   * @req SRD-R-FSD-02
```

```
         */
        log(”TC-mbt-fsd-TR-0006, TC-mbt-fsd-BCS-0004”);
        triggered := true;
      }
      triggered := true;
    }
    /* Handle location ’FSD.Redundancy_Lost’ */
    if ((sut_FSD_currentLocation == 13) and (not triggered))
    {
      if ((not triggered) and ((IOpre_SD1A_state == 2) or (IOpre_SD1B_state == 2)))
      {
        haveDiscreteTrans := true;
        Fire_Detected_expected := true;
        /* New location is ’FSD.Alarm’ */
        sut_FSD_currentLocation := 8;
        /**
         * Model coverage goal : transition coverage
         * Cover transition of component IMR.SystemUnderTest.FSD
         *   FSD.Redundancy_Lost
         *       -- [ ((IMR.SD1A_state == 2) || (IMR.SD1B_state == 2)) ] -->
         *   FSD.Alarm
         *
         * @tag       TC-mbt-fsd-TR-0007 Cover transition Redundancy_Lost --> Alarm
         * @condition Component IMR.SystemUnderTest.FSD
         *            resides in control state IMR.SystemUnderTest.FSD.FSD.Redundancy_Lost
         * @event     Trigger condition for specified transition becomes true
         * @expected  The actions associated with the transition specified above,
         *            and the target state’s entry actions are
         *            performed as specified in the model.
         *
         * @note These checks are performed by the test oracles associated
         *       with component IMR.SystemUnderTest.FSD
         * @req SRD-R-FSD-05
         */
        /**
         * Model coverage goal: basic control state coverage
         *
         * @tag       TC-mbt-fsd-BCS-0001 Cover basic control state Alarm
         * @condition TRUE
         * @event     Component IMR.SystemUnderTest.FSD
         *            reaches basic control state IMR.SystemUnderTest.FSD.FSD.Alarm
         * @expected  The actions associated with the transition entering
         *            this control state, and the control state’s entry actions are
         *            performed as specified in the model.
         *
         * @note These checks are performed by the test oracles associated
         *       with component IMR.SystemUnderTest.FSD
         * @req SRD-R-FSD-03
         */
        log(”TC-mbt-fsd-TR-0007, TC-mbt-fsd-BCS-0001”);
        triggered := true;
      }
      if ((not triggered) and ((IOpre_SD1A_state == 1) and (IOpre_SD1B_state == 1)))
      {
        haveDiscreteTrans := true;
        Maintenance_Required_expected := true;
        /* New location is ’FSD.SD_Failed’ */
        sut_FSD_currentLocation := 14;
        /**
         * Model coverage goal : transition coverage
         * Cover transition of component IMR.SystemUnderTest.FSD
         *   FSD.Redundancy_Lost
         *       -- [ ((IMR.SD1A_state == 1) && (IMR.SD1B_state == 1)) ] -->
         *   FSD.SD_Failed
         *
         * @tag       TC-mbt-fsd-TR-0008 Cover transition Redundancy_Lost --> SD_Failed
         * @condition Component IMR.SystemUnderTest.FSD
         *            resides in control state IMR.SystemUnderTest.FSD.FSD.Redundancy_Lost
         * @event     Trigger condition for specified transition becomes true
         * @expected  The actions associated with the transition specified above,
         *            and the target state’s entry actions are
         *            performed as specified in the model.
         *
         * @note These checks are performed by the test oracles associated
         *       with component IMR.SystemUnderTest.FSD
         * @req SRD-R-FSD-05
```

```
     */
    /**
     * Model coverage goal: basic control state coverage
     *
     * @tag       TC-mbt-fsd-BCS-0004 Cover basic control state SD_Failed
     * @condition TRUE
     * @event     Component IMR.SystemUnderTest.FSD
     *            reaches basic control state IMR.SystemUnderTest.FSD.FSD.SD_Failed
     * @expected  The actions associated with the transition entering
     *            this control state, and the control state’s entry actions are
     *            performed as specified in the model.
     *
     * @note These checks are performed by the test oracles associated
     *       with component IMR.SystemUnderTest.FSD
     * @req SRD-R-FSD-02
     */
    log(”TC-mbt-fsd-TR-0008, TC-mbt-fsd-BCS-0004”);
    triggered := true;
  }
  triggered := true;
}
/* Handle location ’FSD.SD_Failed’ */
if ((sut_FSD_currentLocation == 14) and (not triggered))
{
  if ((not triggered) and ((IOpre_SD1A_state == 0) and (IOpre_SD1B_state == 0)))
  {
    haveDiscreteTrans := true;
    Maintenance_Required_expected := false;
    Fire_Detected_expected := false;
    /* New location is ’FSD.Idle’ */
    sut_FSD_currentLocation := 10;
    /**
     * Model coverage goal : transition coverage
     * Cover transition of component IMR.SystemUnderTest.FSD
     *   FSD.SD_Failed
     *       -- [ ((IMR.SD1A_state == 0) && (IMR.SD1B_state == 0)) ] -->
     *   FSD.Idle
     *
     * @tag       TC-mbt-fsd-TR-0009 Cover transition SD_Failed --> Idle
     * @condition Component IMR.SystemUnderTest.FSD
     *            resides in control state IMR.SystemUnderTest.FSD.FSD.SD_Failed
     * @event     Trigger condition for specified transition becomes true
     * @expected  The actions associated with the transition specified above,
     *            and the target state’s entry actions are
     *            performed as specified in the model.
     *
     * @note These checks are performed by the test oracles associated
     *       with component IMR.SystemUnderTest.FSD
     * @req SRD-R-FSD-02
     */
    /**
     * Model coverage goal: basic control state coverage
     *
     * @tag       TC-mbt-fsd-BCS-0002 Cover basic control state Idle
     * @condition TRUE
     * @event     Component IMR.SystemUnderTest.FSD
     *            reaches basic control state IMR.SystemUnderTest.FSD.FSD.Idle
     * @expected  The actions associated with the transition entering
     *            this control state, and the control state’s entry actions are
     *            performed as specified in the model.
     *
     * @note These checks are performed by the test oracles associated
     *       with component IMR.SystemUnderTest.FSD
     * @req SRD-R-FSD-01
     */
    log(”TC-mbt-fsd-TR-0009, TC-mbt-fsd-BCS-0002”);
    triggered := true;
  }
  triggered := true;
}
/* Handle location ’FSD.SD_Inconsistent’ */
if ((sut_FSD_currentLocation == 16) and (not triggered))
{
  if ((not triggered) and ((IOpre_SD1A_state == 2) and (IOpre_SD1B_state == 2)))
  {
    haveDiscreteTrans := true;
    Fire_Detected_expected := true;
    /* New location is 'FSD.Alarm' */
    sut_FSD_currentLocation := 8;
    /**
     * Model coverage goal : transition coverage
     * Cover transition of component IMR.SystemUnderTest.FSD
     *   FSD.SD_Inconsistent
     *       -- [ ((IMR.SD1A_state == 2) && (IMR.SD1B_state == 2)) ] -->
     *   FSD.Alarm
     *
     * @tag       TC-mbt-fsd-TR-0010 Cover transition SD_Inconsistent --> Alarm
     * @condition Component IMR.SystemUnderTest.FSD
     *            resides in control state IMR.SystemUnderTest.FSD.FSD.SD_Inconsistent
     * @event     Trigger condition for specified transition becomes true
     * @expected  The actions associated with the transition specified above,
     *            and the target state's entry actions are
     *            performed as specified in the model.
     *
     * @note These checks are performed by the test oracles associated
     *       with component IMR.SystemUnderTest.FSD
     * @req SRD-R-FSD-04
     */
    /**
     * Model coverage goal: basic control state coverage
     *
     * @tag       TC-mbt-fsd-BCS-0001 Cover basic control state Alarm
     * @condition TRUE
     * @event     Component IMR.SystemUnderTest.FSD
     *            reaches basic control state IMR.SystemUnderTest.FSD.FSD.Alarm
     * @expected  The actions associated with the transition entering
     *            this control state, and the control state's entry actions are
     *            performed as specified in the model.
     *
     * @note These checks are performed by the test oracles associated
     *       with component IMR.SystemUnderTest.FSD
     * @req SRD-R-FSD-03
     */
    log("TC-mbt-fsd-TR-0010, TC-mbt-fsd-BCS-0001");
    triggered := true;
  }
  if ((not triggered) and ((_timeTick - Statepre_sut_FSD_t) >= 1000))
  {
    haveDiscreteTrans := true;
    Maintenance_Required_expected := true;
    /* New location is 'FSD.SD_Failed' */
    sut_FSD_currentLocation := 14;
    /**
     * Model coverage goal : transition coverage
     * Cover transition of component IMR.SystemUnderTest.FSD
     *   FSD.SD_Inconsistent
     *       -- [ ((_timeTick - IMR.SystemUnderTest.FSD.t) >= 1000) ] -->
     *   FSD.SD_Failed
     *
     * @tag       TC-mbt-fsd-TR-0011 Cover transition SD_Inconsistent --> SD_Failed
     * @condition Component IMR.SystemUnderTest.FSD
     *            resides in control state IMR.SystemUnderTest.FSD.FSD.SD_Inconsistent
     * @event     Trigger condition for specified transition becomes true
     * @expected  The actions associated with the transition specified above,
     *            and the target state's entry actions are
     *            performed as specified in the model.
     *
     * @note These checks are performed by the test oracles associated
     *       with component IMR.SystemUnderTest.FSD
     * @req SRD-R-FSD-04
     */
    /**
     * Model coverage goal: basic control state coverage
     *
     * @tag       TC-mbt-fsd-BCS-0004 Cover basic control state SD_Failed
     * @condition TRUE
     * @event     Component IMR.SystemUnderTest.FSD
     *            reaches basic control state IMR.SystemUnderTest.FSD.FSD.SD_Failed
     * @expected  The actions associated with the transition entering
     *            this control state, and the control state's entry actions are
     *            performed as specified in the model.
     *
     * @note These checks are performed by the test oracles associated
     *       with component IMR.SystemUnderTest.FSD
     * @req SRD-R-FSD-02
     */
    log("TC-mbt-fsd-TR-0011, TC-mbt-fsd-BCS-0004");
    triggered := true;
  }
  triggered := true;
}
/* Perform checks only when system is stable (haveDiscreteTrans == false) */
if (haveDiscreteTrans == false)
{
  /* Checker for signal IMR.Fire_Detected */
  /* Do not perform any checks if IMR.Fire_Detected may still bounce ↩
        ↪ (IMR.Fire_Detected_counter < 0) */
  if (Fire_Detected_counter < 0)
  {
    Fire_Detected_counter := (Fire_Detected_counter + 1);
  }
  if (Fire_Detected_counter >= 0)
  {
    if (Fire_Detected_expectedOld != Fire_Detected_expected)
    {
      Fire_Detected_expectedOld := Fire_Detected_expected;
      Fire_Detected_counter := 100;
      Fire_Detected_passed := false;
      Fire_Detected_failed := false;
    }
    if (Fire_Detected_counter > 0)
    {
      Fire_Detected_counter := (Fire_Detected_counter - 1);
      if (IOpre_Fire_Detected == Fire_Detected_expected)
      {
        Fire_Detected_counter := 0;
        Fire_Detected_passed := true;
        Fire_Detected_failed := false;
        log("(IOpre_Fire_Detected == Fire_Detected_expected)");
        setverdict(pass);
      }
    }
    if (Fire_Detected_counter == 0)
    {
      if ((IOpre_Fire_Detected == Fire_Detected_expected) and (not Fire_Detected_passed))
      {
        Fire_Detected_passed := true;
        Fire_Detected_failed := false;
        log("(IOpre_Fire_Detected == Fire_Detected_expected)");
        setverdict(pass);
      }
      if ((IOpre_Fire_Detected != Fire_Detected_expected) and (not Fire_Detected_failed))
      {
        Fire_Detected_passed := false;
        Fire_Detected_failed := true;
        log("(IOpre_Fire_Detected == Fire_Detected_expected)");
        setverdict(fail);
      }
    }
  }
  /* Checker for signal IMR.Maintenance_Required */
  /* Do not perform any checks if IMR.Maintenance_Required may still bounce ↩
        ↪ (IMR.Maintenance_Required_counter < 0) */
  if (Maintenance_Required_counter < 0)
  {
    Maintenance_Required_counter := (Maintenance_Required_counter + 1);
  }
  if (Maintenance_Required_counter >= 0)
  {
    if (Maintenance_Required_expectedOld != Maintenance_Required_expected)
    {
      Maintenance_Required_expectedOld := Maintenance_Required_expected;
      Maintenance_Required_counter := 100;
      Maintenance_Required_passed := false;
      Maintenance_Required_failed := false;
    }
    if (Maintenance_Required_counter > 0)
    {
          Maintenance_Required_counter := (Maintenance_Required_counter - 1);
          if (IOpre_Maintenance_Required == Maintenance_Required_expected)
          {
            Maintenance_Required_counter := 0;
            Maintenance_Required_passed := true;
            Maintenance_Required_failed := false;
            log("(IOpre_Maintenance_Required == Maintenance_Required_expected)");
            setverdict(pass);
          }
        }
        if (Maintenance_Required_counter == 0)
        {
          if ((IOpre_Maintenance_Required == Maintenance_Required_expected) and (not ↩
               ↪ Maintenance_Required_passed))
          {
            Maintenance_Required_passed := true;
            Maintenance_Required_failed := false;
            log("(IOpre_Maintenance_Required == Maintenance_Required_expected)");
            setverdict(pass);
          }
          if ((IOpre_Maintenance_Required != Maintenance_Required_expected) and (not ↩
               ↪ Maintenance_Required_failed))
          {
            Maintenance_Required_passed := false;
            Maintenance_Required_failed := true;
            log("(IOpre_Maintenance_Required == Maintenance_Required_expected)");
            setverdict(fail);
          }
        }
      }
    }
    if ( true )
    {
      triggered := true;
    }
    cycleCtr := (cycleCtr + 1);
  }
}

function stimulator() runs on MTCType {
  select (stimulator_location) {
    case (0) {
      if (_timeTick >= 0) {
        IOpost_SD1A_state := 1;
        IOpost_SD1B_state := 2;
        stimulator_location := 1;
      }
    }
    case (1) {
      if (_timeTick >= 0) {
        IOpost_SD1A_state := 1;
        IOpost_SD1B_state := 2;
        stimulator_location := 2;
      }
    }
    case (2) {
      if (_timeTick >= 8191) {
        IOpost_SD1A_state := 2;
        IOpost_SD1B_state := 0;
        stimulator_location := 3;
      }
    }
    case (3) {
      if (_timeTick >= 9191) {
        IOpost_SD1A_state := 1;
        IOpost_SD1B_state := 2;
        stimulator_location := 4;
      }
    }
    case (4) {
      if (_timeTick >= 17016) {
        IOpost_SD1A_state := 0;
        IOpost_SD1B_state := 0;
        stimulator_location := 5;
      }
    }
    case (5) {
      if (_timeTick >= 24768) {
        IOpost_SD1A_state := 1;
        IOpost_SD1B_state := 1;
        stimulator_location := 6;
      }
    }
    case (6) {
      if (_timeTick >= 28660) {
        IOpost_SD1A_state := 0;
        IOpost_SD1B_state := 0;
        stimulator_location := 7;
      }
    }
    case (7) {
      if (_timeTick >= 28670) {
        IOpost_SD1A_state := 1;
        stimulator_location := 8;
      }
    }
    case (8) {
      if (_timeTick >= 33671) {
        IOpost_SD1B_state := 1;
        stimulator_location := 9;
      }
    }
    case (9) {
      if (_timeTick >= 42999) {
        IOpost_SD1A_state := 0;
        IOpost_SD1B_state := 0;
        stimulator_location := 10;
      }
    }
    case (10) {
      if (_timeTick >= 52736) {
        IOpost_SD1A_state := 2;
        stimulator_location := 11;
      }
    }
    case (11) {
      if (_timeTick >= 53632) {
        IOpost_SD1B_state := 2;
        stimulator_location := 12;
      }
    }
    case (12) {
      if (_timeTick >= 56464) {
        IOpost_SD1A_state := 0;
        IOpost_SD1B_state := 0;
        stimulator_location := 13;
      }
    }
    case (13) {
      if (_timeTick >= 56490) {
        IOpost_SD1B_state := 1;
        stimulator_location := 14;
      }
    }
    case (14) {
      if (_timeTick >= 56500) {
        IOpost_SD1A_state := 1;
        IOpost_SD1B_state := 2;
        stimulator_location := 15;
      }
    }
    case (15) {
      if (_timeTick >= 61500) {
        stop;
      }
    }
  }
}


function input() runs on MTCType {
  haveDiscreteTrans := false;
```

```
    alt {
      [] Fire_Detected.receive(boolean:?) -> value IOpre_Fire_Detected { repeat; };
      [] Maintenance_Required.receive(boolean:?) -> value IOpre_Maintenance_Required { repeat; };
      [] cycleTimer.timeout { cycleTimer.start; _timeTick := _timeTick + float2int(cycle_period ↩
            ↪ * 1000.0); };
    }
  }

  function output() runs on MTCType {
    if (IOpre_SD1A_state != IOpost_SD1A_state) {
      IOpre_SD1A_state := IOpost_SD1A_state;
      SD1A_state.send(IOpost_SD1A_state);
    }
    if (IOpre_SD1B_state != IOpost_SD1B_state) {
      IOpre_SD1B_state := IOpost_SD1B_state;
      SD1B_state.send(IOpost_SD1B_state);
    }
    Statepre_sut_FSD_t := Statepost_sut_FSD_t;
    Statepre_te__SD1_Sim_t_rep := Statepost_te__SD1_Sim_t_rep;
  }

  testcase Test() runs on MTCType {

    cycleTimer.start;
    while (true) {
      input();
      _ora_FSD();
      stimulator();
      output();
    }
  }
}
```

APPENDIX B

TA Auxiliary Functions

ARINC specification 653 contains formal definitions for API calls. This appendix gives formal definitions of the auxiliary functions supported by the Test Agent. The same style as used in the ARINC 653 API call definitions is applied here.

B.1 AUX_RESERVE_DATA_TABLE_ENTRY

The function AUX_RESERVE_DATA_TABLE_ENTRY shall find a free, empty data table entry and reserve it for use.

```
Procedure AUX_RESERVE_DATA_TABLE_ENTRY
    (LENGTH      : in  NUMERIC;
     INDEX       : out NUMERIC;
     RETURN_CODE : out RETURN_CODE_TYPE) is
error
    when (no more entries available) =>
        RETURN_CODE := NOT_AVAILABLE;
    when (not enough memory available) =>
        RETURN_CODE := INVALID_PARAM;
normal
    set INDEX to next free data table entry;
    initialise data at table index INDEX to 0;
    RETURN_CODE := NO_ERROR;
end AUX_RESERVE_DATA_TABLE_ENTRY;
```

B.2 AUX_CLEAR_DATA_TABLE_ENTRY

The function AUX_CLEAR_DATA_TABLE_ENTRY shall remove an entry from the data table and release its memory.

```
Procedure AUX_CLEAR_DATA_TABLE_ENTRY
    (INDEX       : in  NUMERIC;
     RETURN_CODE : out RETURN_CODE_TYPE) is
error
    when (INDEX >= size of data table) =>
        RETURN_CODE := INVALID_PARAM;
normal
    set data table entry at index INDEX to be unused;
    RETURN_CODE := NO_ERROR;
end AUX_CLEAR_DATA_TABLE_ENTRY;
```

B.3 AUX_GET_DATA_TABLE_ENTRY

The function AUX_GET_DATA_TABLE_ENTRY shall read the data of an entry and send it back to the test environment.

```
Procedure AUX_GET_DATA_TABLE_ENTRY
    (INDEX       : in  NUMERIC;
     LENGTH      : out NUMERIC;
     DATA        : out BYTE_ARRAY;
     RETURN_CODE : out RETURN_CODE_TYPE) is
error
    when (INDEX >= size of data table) =>
        RETURN_CODE := INVALID_PARAM;
normal
    LENGTH := actual length of entry (0 if invalid);
    DATA := data contents of entry;
    RETURN_CODE := NO_ERROR;
end AUX_GET_DATA_TABLE_ENTRY;
```

B.4 AUX_SET_DATA_TABLE_ENTRY

The function AUX_SET_DATA_TABLE_ENTRY shall set the contents of an entry to the data specified by the test environment.

```
Procedure AUX_SET_DATA_TABLE_ENTRY
    (INDEX       : in  NUMERIC;
     LENGTH      : in  NUMERIC;
     DATA        : in  BYTE_ARRAY;
     RETURN_CODE : out RETURN_CODE_TYPE) is
```

```
error
    when (INDEX >= size of data table) =>
        RETURN_CODE := INVALID_PARAM;
    when (not enough memory available) =>
        RETURN_CODE := INVALID_PARAM;
normal
    set size of data table entry at index INDEX
      to LENGTH;
    copy DATA into table entry at index INDEX;
    RETURN_CODE := NO_ERROR;
end AUX_SET_DATA_TABLE_ENTRY;
```

APPENDIX C

TTCN-3 Framework Components

This appendix lists the implementations of several components that make up the TTCN-3 SUT interface of the ITML framework. In particular, it shows the Proxy SUT Adapter, the TACP SUT Adapter, the TACP codec reference implementations, and the TACP protocol handler as described in sections 8.1 to 8.3.

C.1 Proxy SA Implementation

```
package itml;

import java.util.HashMap;
import java.util.LinkedList;
import java.util.ListIterator;

import org.etsi.ttcn.tci.TciCDProvided;
import org.etsi.ttcn.tri.TriAddress;
import org.etsi.ttcn.tri.TriComponentId;
import org.etsi.ttcn.tri.TriMessage;
import org.etsi.ttcn.tri.TriPortId;
import org.etsi.ttcn.tri.TriPortIdList;
import org.etsi.ttcn.tri.TriStatus;
import org.etsi.ttcn.tri.TriTestCaseId;

import com.testingtech.ttcn.logging.RTLoggingConstants;
import com.testingtech.ttcn.tci.codec.SimpleCodec;
import com.testingtech.ttcn.tri.TestAdapter;
import com.testingtech.ttcn.tri.TriStatusImpl;

import de.tu_berlin.cs.uebb.muttcn.runtime.RB;

public class ProxySA extends TestAdapter {
```

```
private static final long serialVersionUID = 42L;

// mapping of port names to sub-SAs
private HashMap<String, TestAdapter> portMap;
private HashMap<String, Integer> portRefCnt;

// list of registered sub-SAs
private LinkedList<TestAdapter> subsaList;

public ProxySA() {
  super();
  portMap = new HashMap<String, TestAdapter>();
  portRefCnt = new HashMap<String, Integer>();
  subsaList = new LinkedList<TestAdapter>();
  subsaList.add(new TACP_SA());
  subsaList.add(new ADSAdapter());
}

public TestAdapter setRB(RB rb) {
  super.setRB(rb);

  ListIterator<TestAdapter> i = subsaList.listIterator();
  while (i.hasNext()) {
    TestAdapter subsa = i.next();
    subsa.setRB(rb);
  }
  return this;
}

// find codec provided by sub-SA for given encoding
public TciCDProvided getCodec(String encodingName) {
  if ((encodingName == null) || encodingName.equals("")) {
    encodingName = "SimpleCodec";
  }

  TciCDProvided codec = super.getCodec(encodingName);

  if (codec != null) {
    return codec;
  }

  if (encodingName.equals("SimpleCodec")) {
    codec = new SimpleCodec(RB);
    codecs.put(encodingName, codec);
  } else {
    ListIterator<TestAdapter> i = subsaList.listIterator();
    while (i.hasNext()) {
      TestAdapter subsa = i.next();
      codec = subsa.getCodec(encodingName);
      if (codec != null) {
        codecs.put(encodingName, codec);
        break;
      }
    }
  }

  if (codec == null) {
    RB.getTciTLProvidedV321TT().tliRT("", System.currentTimeMillis(),
      "", -1, null, RTLoggingConstants.RT_LOG_ERROR,
      "Unknown decoding " + encodingName);
    RB.tciTMProvided.tciError("Unknown decoding " + encodingName);
  }

  return codec;
}

// reset SUT
```

```
@Override
public TriStatus triSAReset() {
  ListIterator<TestAdapter> i = subsaList.listIterator();
  while (i.hasNext()) {
    TestAdapter subsa = i.next();
    subsa.triSAReset();
  }
  portMap.clear();
  portRefCnt.clear();

  return new TriStatusImpl();
}

// begin test case
@Override
public TriStatus triExecuteTestcase(TriTestCaseId testcase,
                  TriPortIdList tsiList) {
  ListIterator<TestAdapter> i = subsaList.listIterator();
  while (i.hasNext()) {
    TestAdapter subsa = i.next();
    subsa.triExecuteTestcase(testcase, tsiList);
  }

  return new TriStatusImpl();
}

// end test case
@Override
public TriStatus triEndTestCase() {
  ListIterator<TestAdapter> i = subsaList.listIterator();
  while (i.hasNext()) {
    TestAdapter subsa = i.next();
    subsa.triEndTestCase();
  }

  return new TriStatusImpl();
}

// find sub-SA responsible for mapping a TSI port
@Override
public TriStatus triMap(final TriPortId compPortId,
            final TriPortId tsiPortId) {
  // already in map?
  String tsiPortName = tsiPortId.toString();
  TestAdapter subsa = portMap.get(tsiPortName);
  if (subsa != null) {
    TriStatus mapStatus = subsa.triMap(compPortId, tsiPortId);
    if (mapStatus.getStatus() == TriStatus.TRI_OK) {
      Integer refCnt = portRefCnt.get(tsiPortName);
      refCnt++;
    }
    return mapStatus;
  } else {
    // not yet mapped: go through list of Sub-SAs
    ListIterator<TestAdapter> i = subsaList.listIterator();
    while (i.hasNext()) {
      subsa = i.next();
      TriStatus mapStatus = subsa.triMap(compPortId, tsiPortId);
      if (mapStatus.getStatus() == TriStatus.TRI_OK) {
        portMap.put(tsiPortName, subsa);
        portRefCnt.put(tsiPortName, 1);
        return mapStatus;
      }
    }
  }

  return new TriStatusImpl("ProxySA: Cannot map port " +
                tsiPortId.getPortName());
  }

  // unmap a TSI port
  @Override
  public TriStatus triUnmap(TriPortId compPortId,
              TriPortId tsiPortId) {
    String tsiPortName = tsiPortId.toString();
    TestAdapter subsa = portMap.get(tsiPortName);
    if (subsa != null) {
      TriStatus mapStatus = subsa.triUnmap(compPortId, tsiPortId);

      Integer refCnt = portRefCnt.get(tsiPortName);
      if (--refCnt == 0) {
        portMap.remove(tsiPortName);
        portRefCnt.remove(tsiPortName);
      }
      return mapStatus;
    }

    return new TriStatusImpl(”ProxySA: triUnmap on unmapped port ” +
                tsiPortId.getPortName());
  }

  // send message over TSI port via sub-SA which mapped the port
  @Override
  public TriStatus triSend(final TriComponentId componentId,
              final TriPortId tsiPortId,
              final TriAddress sutAddress,
              final TriMessage message) {
    TestAdapter subsa = portMap.get(tsiPortId.toString());
    if (subsa != null) {
      return subsa.triSend(componentId, tsiPortId, sutAddress, message);
    }

    return new TriStatusImpl(”ProxySA: triSend on unmapped port ” +
                tsiPortId.getPortName());
  }
}
```

C.2 TACP SA Implementation

```
package itml;

import org.etsi.ttcn.tci.TciCDProvided;
import org.etsi.ttcn.tri.TriAddress;
import org.etsi.ttcn.tri.TriComponentId;
import org.etsi.ttcn.tri.TriMessage;
import org.etsi.ttcn.tri.TriPortId;
import org.etsi.ttcn.tri.TriPortIdList;
import org.etsi.ttcn.tri.TriStatus;
import org.etsi.ttcn.tri.TriTestCaseId;

import com.testingtech.ttcn.tri.TestAdapter;
import com.testingtech.ttcn.tri.TriStatusImpl;

public class TACP_SA extends TestAdapter {
  private static final long serialVersionUID = 42L;

  public TACP_SA() {
    super();
  }

  // dummy implementations: TACP SA does not provide h/w access
  @Override
  public TriStatus triSend(final TriComponentId componentId,
             final TriPortId tsiPortId,
             final TriAddress sutAddress,
             final TriMessage message) {
    return new TriStatusImpl(TriStatus.TRI_ERROR);
  }

  @Override
  public TriStatus triExecuteTestcase(TriTestCaseId testcase,
                  TriPortIdList tsiList) {
    return new TriStatusImpl();
  }

  @Override
  public TriStatus triEndTestCase() {
    return new TriStatusImpl();
  }

  @Override
  public TriStatus triSAReset() {
    return super.triSAReset();
  }

  @Override
  public TriStatus triMap(final TriPortId compPortId,
            final TriPortId tsiPortId) {
    return new TriStatusImpl(TriStatus.TRI_ERROR);
  }

  @Override
  public TriStatus triUnmap(TriPortId compPortId,
             TriPortId tsiPortId) {
    return new TriStatusImpl(TriStatus.TRI_ERROR);
  }

  // provide Codecs for TACP protocol and floats
  @Override
  public TciCDProvided getCodec(String encodingName) {
    TciCDProvided codec = null;

    if (encodingName.equals("TACP_CODEC")) {
      codec = new TACP_CODEC(RB);
    }
    if (encodingName.equals("FLOAT_CODEC")) {
      codec = new FLOAT_CODEC(RB);
    }
    return codec;
  }
}
```

C.3 TACP Codec Implementation

```
package scarlett;

import org.etsi.ttcn.tci.TciCDProvided;
import org.etsi.ttcn.tci.TciTypeClass;
import org.etsi.ttcn.tci.Type;
import org.etsi.ttcn.tci.Value;
import org.etsi.ttcn.tci.RecordValue;
import org.etsi.ttcn.tci.RecordOfValue;
import org.etsi.ttcn.tci.UnionValue;
import org.etsi.ttcn.tci.IntegerValue;
import org.etsi.ttcn.tci.EnumeratedValue;
import org.etsi.ttcn.tci.CharstringValue;
import org.etsi.ttcn.tri.TriMessage;
import com.testingtech.ttcn.tri.TriMessageImpl;
import de.tu_berlin.cs.uebb.muttcn.runtime.RB;

import java.io.ByteArrayOutputStream;
import java.io.DataOutputStream;
import java.io.IOException;
import java.io.UnsupportedEncodingException;
import java.nio.ByteBuffer;


public class TACP_CODEC implements TciCDProvided {

  public TACP_CODEC(RB rb) {
  }

  public TriMessage encode(Value v) {
    //System.out.println("TACP_CODEC.encode");
    ByteArrayOutputStream data = new ByteArrayOutputStream();
    DataOutputStream os = new DataOutputStream(data);

    try {
      encodeValue(os, v);
    } catch (IOException e) {
      e.printStackTrace();
    }

    return new TriMessageImpl(data.toByteArray());
  }

  private void encodeValue(DataOutputStream os, Value v) throws IOException {
    if (v.notPresent()) {
      return;
    }
    Type vType = v.getType();
    switch (vType.getTypeClass()) {
    case TciTypeClass.RECORD:
      encodeRecord(os, (RecordValue)v);
      break;

    case TciTypeClass.RECORD_OF:
      encodeRecordOf(os, (RecordOfValue)v);
      break;

    case TciTypeClass.UNION:
      encodeUnion(os, (UnionValue)v);
      break;

    case TciTypeClass.INTEGER:
      encodeInteger(os, (IntegerValue)v);
      break;

    case TciTypeClass.ENUMERATED:
      encodeEnumerated(os, (EnumeratedValue)v);
      break;

    case TciTypeClass.CHARSTRING:
      encodeCharstring(os, (CharstringValue)v);
      break;

    default:
      System.out.println("TACP_CODEC.encode: Unknown Value type: " + ↩
           ↪ vType.getTypeClass());
    }
  }
```

```
private void encodeRecord(DataOutputStream os, RecordValue v) throws ↩
      ↪ IOException {
  String[] fields = v.getFieldNames();

  for (int i = 0; i < fields.length; i++) {
    encodeValue(os, v.getField(fields[i]));
  }
}

private void encodeRecordOf(DataOutputStream os, RecordOfValue v) throws ↩
      ↪ IOException {
  int len = v.getLength();

  for (int i = v.getOffset(); i < len; i++) {
    encodeValue(os, v.getField(i));
  }
}

private void encodeUnion(DataOutputStream os, UnionValue v) throws IOException {
  String typeName = v.getType().getName();

  if (typeName.equals("Operation")) {
    encodeOperationUnion(os, v);
  }
  else {
    System.out.println("TACP_CODEC.encode: Unknown union type: " + typeName);
  }
}

private void encodeOperationUnion(DataOutputStream os, UnionValue v) throws ↩
      ↪ IOException {
  String variant = v.getPresentVariantName();

  for (int i = 0; i < OperationEnumValues.length; i++) {
    if (variant.equals(OperationEnumValues[i] + "_cmd")) {
      os.writeShort((short)i);
      encodeValue(os, v.getVariant(variant));
      return;
    }
  }
  for (int i = 0; i < AuxFuncEnumValues.length; i++) {
    if (variant.equals(AuxFuncEnumValues[i] + "_cmd")) {
      os.writeShort((short)(i + 1001));
      encodeValue(os, v.getVariant(variant));
      return;
    }
  }
  for (int i = 0; i < ScenarioEnumValues.length; i++) {
    if (variant.equals(ScenarioEnumValues[i] + "_cmd")) {
      os.writeShort((short)(i + 2000));
      encodeValue(os, v.getVariant(variant));
      return;
    }
  }
  System.out.println("TACP_CODEC.encode: Unknown union variant: Operation: " + ↩
        ↪ variant);
}

private void encodeInteger(DataOutputStream os, IntegerValue v) throws ↩
      ↪ IOException {
  int i = v.getInt();
  String typeName = v.getType().getName();

  if (typeName.equals("uint8")) {
    os.writeByte((byte)i);
  }
  else if (typeName.equals("uint16")) {
      os.writeShort((short)i);
    }
    else if (typeName.equals("uint32")) {
      os.writeInt(i);
    }
    else {
      System.out.println("TACP_CODEC.encode: Unknown integer type: " + typeName);
    }
  }

  private void encodeEnumerated(DataOutputStream os, EnumeratedValue v) throws ↩
       ↪ IOException {
    String typeName = v.getType().getName();

    System.out.println("TACP_CODEC.encode: Unknown enum type: " + typeName);
  }

  private void encodeCharstring(DataOutputStream os, CharstringValue v) throws ↩
       ↪ IOException {
    byte[] s = v.getString().getBytes();
    os.writeInt(s.length + 1);
    for (int i = 0; i < s.length; i++) {
      os.writeByte(s[i]);
    }
    os.writeByte(0);

    int padding = (4 - (s.length + 1) % 4) % 4;
    for (int i = 0; i < padding; i++) {
      os.writeByte(0);
    }
  }


  public Value decode(TriMessage msg, Type vType) {
    //System.out.println("TACP_CODEC.decode");
    ByteBuffer buf = ByteBuffer.wrap(msg.getEncodedMessage());
    Value v = vType.newInstance();

    decodeValue(buf, v);

    return v;
  }

  private boolean decodeValue(ByteBuffer buf, Value v) {
    Type vType = v.getType();
    boolean valueRead = false;

    switch (vType.getTypeClass()) {
    case TciTypeClass.RECORD:
      valueRead = decodeRecord(buf, (RecordValue)v);
      break;

    case TciTypeClass.RECORD_OF:
      valueRead = decodeRecordOf(buf, (RecordOfValue)v);
      break;

    case TciTypeClass.UNION:
      valueRead = decodeUnion(buf, (UnionValue)v);
      break;

    case TciTypeClass.INTEGER:
      valueRead = decodeInteger(buf, (IntegerValue)v);
      break;

    case TciTypeClass.ENUMERATED:
      valueRead = decodeEnumerated(buf, (EnumeratedValue)v);
      break;
  case TciTypeClass.CHARSTRING:
    valueRead = decodeCharstring(buf, (CharstringValue)v);
    break;

  default:
    System.out.println(”TACP_CODEC.decode: Unknown Value type: ” + ↩
         ↪ vType.getTypeClass());
  }
  return valueRead;
}

private boolean decodeRecord(ByteBuffer buf, RecordValue v) {
  String[] fields = v.getFieldNames();
  boolean readValues = false;

  for (int i = 0; i < fields.length; i++) {
    Value fv = v.getField(fields[i]).getType().newInstance();
    if (!decodeValue(buf, fv)) {
      v.setFieldOmitted(fields[i]);
    } else {
      v.setField(fields[i], fv);
      readValues = true;
    }
  }
  return readValues || fields.length == 0;
}

private boolean decodeRecordOf(ByteBuffer buf, RecordOfValue v) {
  String typeName = v.getType().getName();

  System.out.println(”TACP_CODEC.decode: Unknown record-of type: ” + typeName);
  return false;
}

private boolean decodeUnion(ByteBuffer buf, UnionValue v) {
  String typeName = v.getType().getName();

  if (typeName.equals(”Operation”)) {
    return decodeOperationUnion(buf, v);
  }
  else {
    System.out.println(”TACP_CODEC.decode: Unknown union type: ” + typeName);
  }
  return false;
}

private boolean decodeOperationUnion(ByteBuffer buf, UnionValue v) {
  if (buf.remaining() >= 2) {
    int op = buf.getShort();
    String variantName;
    boolean readValue;

    if (op >= 0 && op < OperationEnumValues.length) {
      variantName = OperationEnumValues[op] + ”_resp”;
    }
    else if (op >= 1001 && op < AuxFuncEnumValues.length + 1001) {
      variantName = AuxFuncEnumValues[op - 1001] + ”_resp”;
    }
    else if (op >= 2000 && op < ScenarioEnumValues.length + 2000) {
      variantName = ScenarioEnumValues[op - 2000] + ”_resp”;
    }
    else {
      System.out.println(”TACP_CODEC.decode: Unknown value: Operation: ” + op);
      variantName = ”INVALID_OPERATION_resp”;
    }
```

```
      Value vv = v.getVariant(variantName).getType().newInstance();
      readValue = decodeValue(buf, vv);
      v.setVariant(variantName, vv);
      return readValue;
    }
    else {
      return false;
    }
  }

  private boolean decodeInteger(ByteBuffer buf, IntegerValue v) {
    String typeName = v.getType().getName();

    if (typeName.equals(”uint8”)) {
      if (buf.remaining() >= 1) {
        v.setInt((buf.get() + 256) % 256);
        return true;
      }
    }
    else if (typeName.equals(”uint16”)) {
      if (buf.remaining() >= 2) {
        v.setInt(buf.getShort());
        return true;
      }
    }
    else if (typeName.equals(”uint32”)) {
      if (buf.remaining() >= 4) {
        v.setInt(buf.getInt());
        return true;
      }
    }
    else {
      System.out.println(”TACP_CODEC.decode: Unknown integer type: ” + typeName);
    }
    return false;
  }

  private boolean decodeEnumerated(ByteBuffer buf, EnumeratedValue v) {
    String typeName = v.getType().getName();

    System.out.println(”TACP_CODEC.decode: Unknown enumeration type: ” + typeName);
    return false;
  }

  private boolean decodeCharstring(ByteBuffer buf, CharstringValue v) {
    if (buf.remaining() >= 4) {
      int length = buf.getInt();

      if (buf.remaining() >= length) {
        String s = new String();

        if (length - 1 > 0) {
          try {
            s = new String(buf.array(), buf.position(), length - 1, ”ISO-8859-1”);
          } catch (UnsupportedEncodingException e) {
            e.printStackTrace();
          }
          int padding = (4 - length % 4) % 4;
          buf.position(buf.position() + length + padding);
        }
        v.setString(s);
        return true;
      }
    }
    return false;
  }
```

```
private static final String OperationEnumValues[] = {
  ”INVALID_OPERATION”,
  ”API_CLEAR_BLACKBOARD”,
  ”API_CREATE_BLACKBOARD”,
  ”API_CREATE_BUFFER”,
  ”API_CREATE_ERROR_HANDLER”,
  ”API_CREATE_EVENT”,
  ”API_CREATE_PROCESS”,
  ”API_CREATE_QUEUING_PORT”,
  ”API_CREATE_SAMPLING_PORT”,
  ”API_CREATE_SEMAPHORE”,
  ”API_DELAYED_START”,
  ”API_DISPLAY_BLACKBOARD”,
  ”API_GET_BLACKBOARD_ID”,
  ”API_GET_BLACKBOARD_STATUS”,
  ”API_GET_BUFFER_ID”,
  ”API_GET_BUFFER_STATUS”,
  ”API_GET_ERROR_STATUS”,
  ”API_GET_EVENT_ID”,
  ”API_GET_EVENT_STATUS”,
  ”API_GET_MY_ID”,
  ”API_GET_PARTITION_STATUS”,
  ”API_GET_PROCESS_ID”,
  ”API_GET_PROCESS_STATUS”,
  ”API_GET_QUEUING_PORT_ID”,
  ”API_GET_QUEUING_PORT_STATUS”,
  ”API_GET_SAMPLING_PORT_ID”,
  ”API_GET_SAMPLING_PORT_STATUS”,
  ”API_GET_SEMAPHORE_ID”,
  ”API_GET_SEMAPHORE_STATUS”,
  ”API_GET_TIME”,
  ”API_LOCK_PREEMPTION”,
  ”API_PERIODIC_WAIT”,
  ”API_RAISE_APPLICATION_ERROR”,
  ”API_READ_BLACKBOARD”,
  ”API_READ_SAMPLING_MESSAGE”,
  ”API_RECEIVE_BUFFER”,
  ”API_RECEIVE_QUEUING_MESSAGE”,
  ”API_REPLENISH”,
  ”API_REPORT_APPLICATION_MESSAGE”,
  ”API_RESET_EVENT”,
  ”API_RESUME”,
  ”API_SEND_BUFFER”,
  ”API_SEND_QUEUING_MESSAGE”,
  ”API_SET_EVENT”,
  ”API_SET_PARTITION_MODE”,
  ”API_SET_PRIORITY”,
  ”API_SIGNAL_SEMAPHORE”,
  ”API_START”,
  ”API_STOP”,
  ”API_STOP_SELF”,
  ”API_SUSPEND”,
  ”API_SUSPEND_SELF”,
  ”API_TIMED_WAIT”,
  ”API_UNLOCK_PREEMPTION”,
  ”API_WAIT_EVENT”,
  ”API_WAIT_SEMAPHORE”,
  ”API_WRITE_SAMPLING_MESSAGE”,
};

private static final String AuxFuncEnumValues[] = {
  ”AUX_GET_DATA_TABLE_ENTRY”,
  ”AUX_SET_DATA_TABLE_ENTRY”,
  ”AUX_RESERVE_DATA_TABLE_ENTRY”,
  ”AUX_CLEAR_DATA_TABLE_ENTRY”,
};
```

```
  private static final String ScenarioEnumValues[] = {
    ”SCE_OPEN_SIGNAL_PORTS”,
    ”SCE_READ_SIGNAL”,
    ”SCE_WRITE_SIGNAL”,
    ”SCE_STEPPER_INIT”,
    ”SCE_STEPPER_SET”,
    ”SCE_STEPPER_GET”,
  };
}
```

C.4 Float Codec Implementation

```
package itml;

import org.etsi.ttcn.tci.TciCDProvided;
import org.etsi.ttcn.tci.TciCDRequired;
import org.etsi.ttcn.tci.TciTypeClass;
import org.etsi.ttcn.tci.Type;
import org.etsi.ttcn.tci.Value;
import org.etsi.ttcn.tci.FloatValue;
import org.etsi.ttcn.tri.TriMessage;

import com.testingtech.ttcn.tri.TriMessageImpl;
import de.tu_berlin.cs.uebb.muttcn.runtime.RB;

import java.nio.ByteBuffer;

public class FLOAT_CODEC implements TciCDProvided {

  private TciCDRequired mTciCDReq;

  public FLOAT_CODEC(RB rb) {
    mTciCDReq = rb.getTciCDRequired();
  }

  // encode a float value as IEEE 754 binary32
  public TriMessage encode(Value v) {
    ByteBuffer buf = ByteBuffer.allocate(4);
    Type vType = v.getType();

    // check type
    if (vType.getTypeClass() == TciTypeClass.FLOAT) {
      FloatValue fv = (FloatValue)v;
      float f = fv.getFloat();

      buf.putFloat(f);
    } else {
      System.out.println(”FLOAT_CODEC.encode: wrong type class”);
    }

    return new TriMessageImpl(buf.array());
  }

  // decode a float value from IEEE 754 binary32
  public Value decode(TriMessage msg, Type type) {
    if (type.getTypeClass() == TciTypeClass.FLOAT) {
      ByteBuffer buf = ByteBuffer.wrap(msg.getEncodedMessage());

      // check message size
      if (buf.limit() >= 4) {
        float f = buf.getFloat();
        FloatValue fv = (FloatValue)mTciCDReq.
          getFloat().newInstance();
        fv.setFloat(f);
        return fv;
      }
    } else {
      System.out.println(”FLOAT_CODEC.decode: wrong type class”);
    }

    return mTciCDReq.getFloat().newInstance();
  }
}
```

C.5 Common TTCN-3 Type Definitions

```
module types {

    type integer uint8 (0..255);
    type integer uint16 (0..65535);
    type integer uint32 (-2147483648..2147483647);
    type float float32 with { encode ”FLOAT_CODEC” };

    type record INVALID_OPERATION_cmd_t {
    };

    type record INVALID_OPERATION_resp_t {
    };

    type record API_CLEAR_BLACKBOARD_cmd_t {
        uint32 blackboard_id
    };

    type record API_CLEAR_BLACKBOARD_resp_t {
        uint32 return_code
    };

    type record API_CREATE_BLACKBOARD_cmd_t {
        uint32 name,
        uint32 max_msg_size
    };

    type record API_CREATE_BLACKBOARD_resp_t {
        uint32 blackboard_id,
        uint32 return_code
    };

    type record API_CREATE_BUFFER_cmd_t {
        uint32 name,
        uint32 max_msg_size,
        uint32 max_nb_msg,
        uint32 queuing_disc
    };

    type record API_CREATE_BUFFER_resp_t {
        uint32 buffer_id,
        uint32 return_code
    };

    type record API_CREATE_ERROR_HANDLER_cmd_t {
        uint32 entry_point,
        uint32 stack_size
    };

    type record API_CREATE_ERROR_HANDLER_resp_t {
        uint32 return_code
    };

    type record API_CREATE_EVENT_cmd_t {
        uint32 name
    };

    type record API_CREATE_EVENT_resp_t {
        uint32 event_id,
        uint32 return_code
    };

    type record API_CREATE_PROCESS_cmd_t {
        uint32 name,
        uint32 entry_point,
        uint32 stack_size,
        uint32 base_priority,
        uint32 period,
        uint32 time_capacity,
        uint32 deadline
    };

    type record API_CREATE_PROCESS_resp_t {
        uint32 process_id,
        uint32 return_code
    };

    type record API_CREATE_QUEUING_PORT_cmd_t {
        uint32 name,
        uint32 max_msg_size,
        uint32 max_nb_msg,
        uint32 port_direction,
        uint32 queuing_disc
    };

    type record API_CREATE_QUEUING_PORT_resp_t {
        uint32 port_id,
        uint32 return_code
    };

    type record API_CREATE_SAMPLING_PORT_cmd_t {
        uint32 name,
        uint32 max_msg_size,
        uint32 port_direction,
        uint32 refresh_period
    };

    type record API_CREATE_SAMPLING_PORT_resp_t {
        uint32 port_id,
        uint32 return_code
    };

    type record API_CREATE_SEMAPHORE_cmd_t {
        uint32 name,
        uint32 current_value,
        uint32 max_value,
        uint32 queuing_disc
    };

    type record API_CREATE_SEMAPHORE_resp_t {
        uint32 semaphore_id,
        uint32 return_code
    };

    type record API_DELAYED_START_cmd_t {
        uint32 process_id,
        uint32 delay_time
    };

    type record API_DELAYED_START_resp_t {
        uint32 return_code
```

```
};

type record API_DISPLAY_BLACKBOARD_cmd_t {
    uint32 blackboard_id,
    uint32 msg_addr,
    uint32 msg_length
};

type record API_DISPLAY_BLACKBOARD_resp_t {
    uint32 return_code
};

type record API_GET_BLACKBOARD_ID_cmd_t {
    uint32 name
};

type record API_GET_BLACKBOARD_ID_resp_t {
    uint32 blackboard_id,
    uint32 return_code
};

type record API_GET_BLACKBOARD_STATUS_cmd_t {
    uint32 blackboard_id
};

type record API_GET_BLACKBOARD_STATUS_resp_t {
    uint32 empty_indicator,
    uint32 max_msg_size,
    uint32 waiting_processes,
    uint32 return_code
};

type record API_GET_BUFFER_ID_cmd_t {
    uint32 name
};

type record API_GET_BUFFER_ID_resp_t {
    uint32 buffer_id,
    uint32 return_code
};

type record API_GET_BUFFER_STATUS_cmd_t {
    uint32 buffer_id
};

type record API_GET_BUFFER_STATUS_resp_t {
    uint32 nb_msg,
    uint32 max_nb_msg,
    uint32 max_msg_size,
    uint32 waiting_processes,
    uint32 return_code
};

type record API_GET_ERROR_STATUS_cmd_t {
};

type record API_GET_ERROR_STATUS_resp_t {
    //TODO
    uint32 return_code
};

type record API_GET_EVENT_ID_cmd_t {
    uint32 name
};

type record API_GET_EVENT_ID_resp_t {
    uint32 event_id,
        uint32 return_code
    };

    type record API_GET_EVENT_STATUS_cmd_t {
        uint32 event_id
    };

    type record API_GET_EVENT_STATUS_resp_t {
        uint32 event_state,
        uint32 waiting_processes,
        uint32 return_code
    };

    type record API_GET_MY_ID_cmd_t {
    };

    type record API_GET_MY_ID_resp_t {
        uint32 process_id,
        uint32 return_code
    };

    type record API_GET_PARTITION_STATUS_cmd_t {
    };

    type record API_GET_PARTITION_STATUS_resp_t {
        uint32 id,
        uint32 period,
        uint32 duration,
        uint32 lock_level,
        uint32 operating_mode,
        uint32 start_condition,
        uint32 return_code
    };

    type record API_GET_PROCESS_ID_cmd_t {
        uint32 name
    };

    type record API_GET_PROCESS_ID_resp_t {
        uint32 process_id,
        uint32 return_code
    };

    type record API_GET_PROCESS_STATUS_cmd_t {
        uint32 process_id
    };

    type record API_GET_PROCESS_STATUS_resp_t {
        //TODO
        uint32 return_code
    };

    type record API_GET_QUEUING_PORT_ID_cmd_t {
        uint32 name
    };

    type record API_GET_QUEUING_PORT_ID_resp_t {
        uint32 port_id,
        uint32 return_code
    };

    type record API_GET_QUEUING_PORT_STATUS_cmd_t {
        uint32 port_id
    };

    type record API_GET_QUEUING_PORT_STATUS_resp_t {
        //TODO
    uint32 return_code
};

type record API_GET_SAMPLING_PORT_ID_cmd_t {
    uint32 name
};

type record API_GET_SAMPLING_PORT_ID_resp_t {
    uint32 port_id,
    uint32 return_code
};

type record API_GET_SAMPLING_PORT_STATUS_cmd_t {
    uint32 port_id
};

type record API_GET_SAMPLING_PORT_STATUS_resp_t {
    //TODO
    uint32 return_code
};

type record API_GET_SEMAPHORE_ID_cmd_t {
    uint32 name
};

type record API_GET_SEMAPHORE_ID_resp_t {
    uint32 semaphore_id,
    uint32 return_code
};

type record API_GET_SEMAPHORE_STATUS_cmd_t {
    uint32 semaphore_id
};

type record API_GET_SEMAPHORE_STATUS_resp_t {
    //TODO
    uint32 return_code
};

type record API_GET_TIME_cmd_t {
};

type record API_GET_TIME_resp_t {
    uint32 system_time,
    uint32 return_code
};

type record API_LOCK_PREEMPTION_cmd_t {
};

type record API_LOCK_PREEMPTION_resp_t {
    uint32 lock_level,
    uint32 return_code
};

type record API_PERIODIC_WAIT_cmd_t {
};

type record API_PERIODIC_WAIT_resp_t {
    uint32 return_code
};

type record API_RAISE_APPLICATION_ERROR_cmd_t {
    //TODO
};

type record API_RAISE_APPLICATION_ERROR_resp_t {
    uint32 return_code
};

type record API_READ_BLACKBOARD_cmd_t {
    uint32 blackboard_id,
    uint32 time_out,
    uint32 msg_addr
};

type record API_READ_BLACKBOARD_resp_t {
    uint32 msg_length,
    uint32 return_code
};

type record API_READ_SAMPLING_MESSAGE_cmd_t {
    uint32 port_id,
    uint32 msg_addr
};

type record API_READ_SAMPLING_MESSAGE_resp_t {
    uint32 msg_length,
    uint32 validity,
    uint32 return_code
};

type record API_RECEIVE_BUFFER_cmd_t {
    uint32 buffer_id,
    uint32 time_out,
    uint32 msg_addr
};

type record API_RECEIVE_BUFFER_resp_t {
    uint32 msg_length,
    uint32 return_code
};

type record API_RECEIVE_QUEUING_MESSAGE_cmd_t {
    uint32 port_id,
    uint32 time_out,
    uint32 msg_addr
};

type record API_RECEIVE_QUEUING_MESSAGE_resp_t {
    uint32 msg_length,
    uint32 return_code
};

type record API_REPLENISH_cmd_t {
    uint32 budget_time
};

type record API_REPLENISH_resp_t {
    uint32 return_code
};

type record API_REPORT_APPLICATION_MESSAGE_cmd_t {
    //TODO
};

type record API_REPORT_APPLICATION_MESSAGE_resp_t {
    uint32 return_code
};

type record API_RESET_EVENT_cmd_t {
    uint32 event_id
};
```

```
type record API_RESET_EVENT_resp_t {
    uint32 return_code
};

type record API_RESUME_cmd_t {
    uint32 process_id
};

type record API_RESUME_resp_t {
    uint32 return_code
};

type record API_SEND_BUFFER_cmd_t {
    uint32 buffer_id,
    uint32 msg_addr,
    uint32 msg_length,
    uint32 time_out
};

type record API_SEND_BUFFER_resp_t {
    uint32 return_code
};

type record API_SEND_QUEUING_MESSAGE_cmd_t {
    uint32 port_id,
    uint32 msg_addr,
    uint32 msg_length,
    uint32 time_out
};

type record API_SEND_QUEUING_MESSAGE_resp_t {
    uint32 return_code
};

type record API_SET_EVENT_cmd_t {
    uint32 event_id
};

type record API_SET_EVENT_resp_t {
    uint32 return_code
};

type record API_SET_PARTITION_MODE_cmd_t {
    uint32 operating_mode
};

type record API_SET_PARTITION_MODE_resp_t {
    uint32 return_code
};

type record API_SET_PRIORITY_cmd_t {
    uint32 process_id,
    uint32 priority
};

type record API_SET_PRIORITY_resp_t {
    uint32 return_code
};

type record API_SIGNAL_SEMAPHORE_cmd_t {
    uint32 semaphore_id
};

type record API_SIGNAL_SEMAPHORE_resp_t {
    uint32 return_code
};
```

```
type record API_START_cmd_t {
    uint32 process_id
};

type record API_START_resp_t {
    uint32 return_code
};

type record API_STOP_cmd_t {
    uint32 process_id
};

type record API_STOP_resp_t {
    uint32 return_code
};

type record API_STOP_SELF_cmd_t {
};

type record API_STOP_SELF_resp_t {
};

type record API_SUSPEND_cmd_t {
    uint32 process_id
};

type record API_SUSPEND_resp_t {
    uint32 return_code
};

type record API_SUSPEND_SELF_cmd_t {
    uint32 time_out
};

type record API_SUSPEND_SELF_resp_t {
    uint32 return_code
};

type record API_TIMED_WAIT_cmd_t {
    uint32 delay_time
};

type record API_TIMED_WAIT_resp_t {
    uint32 return_code
};

type record API_UNLOCK_PREEMPTION_cmd_t {
};

type record API_UNLOCK_PREEMPTION_resp_t {
    uint32 lock_level,
    uint32 return_code
};

type record API_WAIT_EVENT_cmd_t {
    uint32 event_id,
    uint32 time_out
};

type record API_WAIT_EVENT_resp_t {
    uint32 return_code
};

type record API_WAIT_SEMAPHORE_cmd_t {
    uint32 semaphore_id,
    uint32 time_out
};
```

```
type record API_WAIT_SEMAPHORE_resp_t {
    uint32 return_code
};

type record API_WRITE_SAMPLING_MESSAGE_cmd_t {
    uint32 port_id,
    uint32 msg_addr,
    uint32 msg_length
};

type record API_WRITE_SAMPLING_MESSAGE_resp_t {
    uint32 return_code
};

type record AUX_GET_DATA_TABLE_ENTRY_cmd_t {
    uint32 idx
};

type record AUX_GET_DATA_TABLE_ENTRY_resp_t {
    uint32 return_code,
    charstring data
};

type record AUX_SET_DATA_TABLE_ENTRY_cmd_t {
    uint32 idx,
    charstring data
};

type record AUX_SET_DATA_TABLE_ENTRY_resp_t {
    uint32 return_code
};

type record AUX_RESERVE_DATA_TABLE_ENTRY_cmd_t {
    //TODO
};

type record AUX_RESERVE_DATA_TABLE_ENTRY_resp_t {
    uint32 return_code
};

type record AUX_CLEAR_DATA_TABLE_ENTRY_cmd_t {
    uint32 idx
};

type record AUX_CLEAR_DATA_TABLE_ENTRY_resp_t {
    uint32 return_code
};

type record SCE_OPEN_SIGNAL_PORTS_cmd_t {
};

type record SCE_OPEN_SIGNAL_PORTS_resp_t {
    uint32 succeeded,
    uint32 failed,
    uint32 return_code
};

type record SCE_READ_SIGNAL_cmd_t {
    uint32 idx
};

type record SCE_READ_SIGNAL_resp_t {
    uint32 signal_value,
    uint32 fs,
    uint32 return_code
};
```

```
type record SCE_WRITE_SIGNAL_cmd_t {
    uint32 idx,
    uint32 signal_value,
    uint32 fs
};

type record SCE_WRITE_SIGNAL_resp_t {
    uint32 return_code
};

type record SCE_STEPPER_INIT_cmd_t {
    uint32 idx
};

type record SCE_STEPPER_INIT_resp_t {
    uint32 return_code
};

type record SCE_STEPPER_SET_cmd_t {
    uint32 idx,
    uint32 cmd,
    uint32 angle
};

type record SCE_STEPPER_SET_resp_t {
    uint32 return_code
};

type record SCE_STEPPER_GET_cmd_t {
    uint32 idx
};

type record SCE_STEPPER_GET_resp_t {
    uint32 status,
    uint32 angle,
    uint32 return_code
};



type union Operation {
    INVALID_OPERATION_cmd_t  INVALID_OPERATION_cmd,
    INVALID_OPERATION_resp_t INVALID_OPERATION_resp,

    API_CLEAR_BLACKBOARD_cmd_t  API_CLEAR_BLACKBOARD_cmd,
    API_CLEAR_BLACKBOARD_resp_t API_CLEAR_BLACKBOARD_resp,

    API_CREATE_BLACKBOARD_cmd_t  API_CREATE_BLACKBOARD_cmd,
    API_CREATE_BLACKBOARD_resp_t API_CREATE_BLACKBOARD_resp,

    API_CREATE_BUFFER_cmd_t  API_CREATE_BUFFER_cmd,
    API_CREATE_BUFFER_resp_t API_CREATE_BUFFER_resp,

    API_CREATE_ERROR_HANDLER_cmd_t  API_CREATE_ERROR_HANDLER_cmd,
    API_CREATE_ERROR_HANDLER_resp_t API_CREATE_ERROR_HANDLER_resp,

    API_CREATE_EVENT_cmd_t  API_CREATE_EVENT_cmd,
    API_CREATE_EVENT_resp_t API_CREATE_EVENT_resp,

    API_CREATE_PROCESS_cmd_t  API_CREATE_PROCESS_cmd,
    API_CREATE_PROCESS_resp_t API_CREATE_PROCESS_resp,

    API_CREATE_QUEUING_PORT_cmd_t  API_CREATE_QUEUING_PORT_cmd,
    API_CREATE_QUEUING_PORT_resp_t API_CREATE_QUEUING_PORT_resp,

    API_CREATE_SAMPLING_PORT_cmd_t  API_CREATE_SAMPLING_PORT_cmd,
API_CREATE_SAMPLING_PORT_resp_t API_CREATE_SAMPLING_PORT_resp,

API_CREATE_SEMAPHORE_cmd_t  API_CREATE_SEMAPHORE_cmd,
API_CREATE_SEMAPHORE_resp_t API_CREATE_SEMAPHORE_resp,

API_DELAYED_START_cmd_t  API_DELAYED_START_cmd,
API_DELAYED_START_resp_t API_DELAYED_START_resp,

API_DISPLAY_BLACKBOARD_cmd_t  API_DISPLAY_BLACKBOARD_cmd,
API_DISPLAY_BLACKBOARD_resp_t API_DISPLAY_BLACKBOARD_resp,

API_GET_BLACKBOARD_ID_cmd_t  API_GET_BLACKBOARD_ID_cmd,
API_GET_BLACKBOARD_ID_resp_t API_GET_BLACKBOARD_ID_resp,

API_GET_BLACKBOARD_STATUS_cmd_t  API_GET_BLACKBOARD_STATUS_cmd,
API_GET_BLACKBOARD_STATUS_resp_t API_GET_BLACKBOARD_STATUS_resp,

API_GET_BUFFER_ID_cmd_t  API_GET_BUFFER_ID_cmd,
API_GET_BUFFER_ID_resp_t API_GET_BUFFER_ID_resp,

API_GET_BUFFER_STATUS_cmd_t  API_GET_BUFFER_STATUS_cmd,
API_GET_BUFFER_STATUS_resp_t API_GET_BUFFER_STATUS_resp,

API_GET_ERROR_STATUS_cmd_t  API_GET_ERROR_STATUS_cmd,
API_GET_ERROR_STATUS_resp_t API_GET_ERROR_STATUS_resp,

API_GET_EVENT_ID_cmd_t  API_GET_EVENT_ID_cmd,
API_GET_EVENT_ID_resp_t API_GET_EVENT_ID_resp,

API_GET_EVENT_STATUS_cmd_t  API_GET_EVENT_STATUS_cmd,
API_GET_EVENT_STATUS_resp_t API_GET_EVENT_STATUS_resp,

API_GET_MY_ID_cmd_t  API_GET_MY_ID_cmd,
API_GET_MY_ID_resp_t API_GET_MY_ID_resp,

API_GET_PARTITION_STATUS_cmd_t  API_GET_PARTITION_STATUS_cmd,
API_GET_PARTITION_STATUS_resp_t API_GET_PARTITION_STATUS_resp,

API_GET_PROCESS_ID_cmd_t  API_GET_PROCESS_ID_cmd,
API_GET_PROCESS_ID_resp_t API_GET_PROCESS_ID_resp,

API_GET_PROCESS_STATUS_cmd_t  API_GET_PROCESS_STATUS_cmd,
API_GET_PROCESS_STATUS_resp_t API_GET_PROCESS_STATUS_resp,

API_GET_QUEUING_PORT_ID_cmd_t  API_GET_QUEUING_PORT_ID_cmd,
API_GET_QUEUING_PORT_ID_resp_t API_GET_QUEUING_PORT_ID_resp,

API_GET_QUEUING_PORT_STATUS_cmd_t  API_GET_QUEUING_PORT_STATUS_cmd,
API_GET_QUEUING_PORT_STATUS_resp_t API_GET_QUEUING_PORT_STATUS_resp,

API_GET_SAMPLING_PORT_ID_cmd_t  API_GET_SAMPLING_PORT_ID_cmd,
API_GET_SAMPLING_PORT_ID_resp_t API_GET_SAMPLING_PORT_ID_resp,

API_GET_SAMPLING_PORT_STATUS_cmd_t  API_GET_SAMPLING_PORT_STATUS_cmd,
API_GET_SAMPLING_PORT_STATUS_resp_t API_GET_SAMPLING_PORT_STATUS_resp,

API_GET_SEMAPHORE_ID_cmd_t  API_GET_SEMAPHORE_ID_cmd,
API_GET_SEMAPHORE_ID_resp_t API_GET_SEMAPHORE_ID_resp,

API_GET_SEMAPHORE_STATUS_cmd_t  API_GET_SEMAPHORE_STATUS_cmd,
API_GET_SEMAPHORE_STATUS_resp_t API_GET_SEMAPHORE_STATUS_resp,

API_GET_TIME_cmd_t  API_GET_TIME_cmd,
API_GET_TIME_resp_t API_GET_TIME_resp,

API_LOCK_PREEMPTION_cmd_t  API_LOCK_PREEMPTION_cmd,
API_LOCK_PREEMPTION_resp_t API_LOCK_PREEMPTION_resp,
API_PERIODIC_WAIT_cmd_t  API_PERIODIC_WAIT_cmd,
API_PERIODIC_WAIT_resp_t API_PERIODIC_WAIT_resp,

API_RAISE_APPLICATION_ERROR_cmd_t  API_RAISE_APPLICATION_ERROR_cmd,
API_RAISE_APPLICATION_ERROR_resp_t API_RAISE_APPLICATION_ERROR_resp,

API_READ_BLACKBOARD_cmd_t  API_READ_BLACKBOARD_cmd,
API_READ_BLACKBOARD_resp_t API_READ_BLACKBOARD_resp,

API_READ_SAMPLING_MESSAGE_cmd_t  API_READ_SAMPLING_MESSAGE_cmd,
API_READ_SAMPLING_MESSAGE_resp_t API_READ_SAMPLING_MESSAGE_resp,

API_RECEIVE_BUFFER_cmd_t  API_RECEIVE_BUFFER_cmd,
API_RECEIVE_BUFFER_resp_t API_RECEIVE_BUFFER_resp,

API_RECEIVE_QUEUING_MESSAGE_cmd_t  API_RECEIVE_QUEUING_MESSAGE_cmd,
API_RECEIVE_QUEUING_MESSAGE_resp_t API_RECEIVE_QUEUING_MESSAGE_resp,

API_REPLENISH_cmd_t  API_REPLENISH_cmd,
API_REPLENISH_resp_t API_REPLENISH_resp,

API_REPORT_APPLICATION_MESSAGE_cmd_t  API_REPORT_APPLICATION_MESSAGE_cmd,
API_REPORT_APPLICATION_MESSAGE_resp_t API_REPORT_APPLICATION_MESSAGE_resp,

API_RESET_EVENT_cmd_t  API_RESET_EVENT_cmd,
API_RESET_EVENT_resp_t API_RESET_EVENT_resp,

API_RESUME_cmd_t  API_RESUME_cmd,
API_RESUME_resp_t API_RESUME_resp,

API_SEND_BUFFER_cmd_t  API_SEND_BUFFER_cmd,
API_SEND_BUFFER_resp_t API_SEND_BUFFER_resp,

API_SEND_QUEUING_MESSAGE_cmd_t  API_SEND_QUEUING_MESSAGE_cmd,
API_SEND_QUEUING_MESSAGE_resp_t API_SEND_QUEUING_MESSAGE_resp,

API_SET_EVENT_cmd_t  API_SET_EVENT_cmd,
API_SET_EVENT_resp_t API_SET_EVENT_resp,

API_SET_PARTITION_MODE_cmd_t  API_SET_PARTITION_MODE_cmd,
API_SET_PARTITION_MODE_resp_t API_SET_PARTITION_MODE_resp,

API_SET_PRIORITY_cmd_t  API_SET_PRIORITY_cmd,
API_SET_PRIORITY_resp_t API_SET_PRIORITY_resp,

API_SIGNAL_SEMAPHORE_cmd_t  API_SIGNAL_SEMAPHORE_cmd,
API_SIGNAL_SEMAPHORE_resp_t API_SIGNAL_SEMAPHORE_resp,

API_START_cmd_t  API_START_cmd,
API_START_resp_t API_START_resp,

API_STOP_cmd_t  API_STOP_cmd,
API_STOP_resp_t API_STOP_resp,

API_STOP_SELF_cmd_t  API_STOP_SELF_cmd,
API_STOP_SELF_resp_t API_STOP_SELF_resp,

API_SUSPEND_cmd_t  API_SUSPEND_cmd,
API_SUSPEND_resp_t API_SUSPEND_resp,

API_SUSPEND_SELF_cmd_t  API_SUSPEND_SELF_cmd,
API_SUSPEND_SELF_resp_t API_SUSPEND_SELF_resp,

API_TIMED_WAIT_cmd_t  API_TIMED_WAIT_cmd,
API_TIMED_WAIT_resp_t API_TIMED_WAIT_resp,
    API_UNLOCK_PREEMPTION_cmd_t  API_UNLOCK_PREEMPTION_cmd,
    API_UNLOCK_PREEMPTION_resp_t API_UNLOCK_PREEMPTION_resp,

    API_WAIT_EVENT_cmd_t  API_WAIT_EVENT_cmd,
    API_WAIT_EVENT_resp_t API_WAIT_EVENT_resp,

    API_WAIT_SEMAPHORE_cmd_t  API_WAIT_SEMAPHORE_cmd,
    API_WAIT_SEMAPHORE_resp_t API_WAIT_SEMAPHORE_resp,

    API_WRITE_SAMPLING_MESSAGE_cmd_t  API_WRITE_SAMPLING_MESSAGE_cmd,
    API_WRITE_SAMPLING_MESSAGE_resp_t API_WRITE_SAMPLING_MESSAGE_resp,

    AUX_GET_DATA_TABLE_ENTRY_cmd_t  AUX_GET_DATA_TABLE_ENTRY_cmd,
    AUX_GET_DATA_TABLE_ENTRY_resp_t AUX_GET_DATA_TABLE_ENTRY_resp,

    AUX_SET_DATA_TABLE_ENTRY_cmd_t  AUX_SET_DATA_TABLE_ENTRY_cmd,
    AUX_SET_DATA_TABLE_ENTRY_resp_t AUX_SET_DATA_TABLE_ENTRY_resp,

    AUX_RESERVE_DATA_TABLE_ENTRY_cmd_t  AUX_RESERVE_DATA_TABLE_ENTRY_cmd,
    AUX_RESERVE_DATA_TABLE_ENTRY_resp_t AUX_RESERVE_DATA_TABLE_ENTRY_resp,

    AUX_CLEAR_DATA_TABLE_ENTRY_cmd_t  AUX_CLEAR_DATA_TABLE_ENTRY_cmd,
    AUX_CLEAR_DATA_TABLE_ENTRY_resp_t AUX_CLEAR_DATA_TABLE_ENTRY_resp,

    SCE_OPEN_SIGNAL_PORTS_cmd_t  SCE_OPEN_SIGNAL_PORTS_cmd,
    SCE_OPEN_SIGNAL_PORTS_resp_t SCE_OPEN_SIGNAL_PORTS_resp,

    SCE_READ_SIGNAL_cmd_t  SCE_READ_SIGNAL_cmd,
    SCE_READ_SIGNAL_resp_t SCE_READ_SIGNAL_resp,

    SCE_WRITE_SIGNAL_cmd_t  SCE_WRITE_SIGNAL_cmd,
    SCE_WRITE_SIGNAL_resp_t SCE_WRITE_SIGNAL_resp,

    SCE_STEPPER_INIT_cmd_t  SCE_STEPPER_INIT_cmd,
    SCE_STEPPER_INIT_resp_t SCE_STEPPER_INIT_resp,

    SCE_STEPPER_SET_cmd_t  SCE_STEPPER_SET_cmd,
    SCE_STEPPER_SET_resp_t SCE_STEPPER_SET_resp,

    SCE_STEPPER_GET_cmd_t  SCE_STEPPER_GET_cmd,
    SCE_STEPPER_GET_resp_t SCE_STEPPER_GET_resp
};

type record tacp_l1_pdu {
    uint8 seq_no,
    uint8 ack_no,
    uint8 flags,
    uint8 spare,
    tacp_l2_pdu l2 optional
} with {
    encode ”TACP_CODEC”
};

type record tacp_l2_pdu {
    uint8 partition,
    uint8 process_idx,
    Operation operation
};

signature set_process_idx(in uint8 new_proc_idx);

// API calls
signature CLEAR_BLACKBOARD(in uint32 blackboard_id,
                           out uint32 return_code);

signature CREATE_BLACKBOARD(in uint32 name,
                            in uint32 max_msg_size,
                                out uint32 blackboard_id,
                                out uint32 return_code);

    signature CREATE_BUFFER(in uint32 name,
                            in uint32 max_msg_size,
                            in uint32 max_nb_msg,
                            in uint32 queuing_disc,
                            out uint32 buffer_id,
                            out uint32 return_code);

    signature CREATE_EVENT(in uint32 name,
                           out uint32 event_id,
                           out uint32 return_code);

    signature CREATE_PROCESS(in uint32 name,
                             in uint32 entry_point,
                             in uint32 stack_size,
                             in uint32 base_priority,
                             in uint32 period,
                             in uint32 time_capacity,
                             in uint32 deadline,
                             out uint32 process_id,
                             out uint32 return_code);

    signature CREATE_QUEUING_PORT(in uint32 name,
                                  in uint32 max_msg_size,
                                  in uint32 max_nb_msg,
                                  in uint32 port_direction,
                                  in uint32 queuing_disc,
                                  out uint32 port_id,
                                  out uint32 return_code);

    signature CREATE_SAMPLING_PORT(in uint32 name,
                                   in uint32 max_msg_size,
                                   in uint32 port_direction,
                                   in uint32 refresh_period,
                                   out uint32 port_id,
                                   out uint32 return_code);

    signature CREATE_SEMAPHORE(in uint32 name,
                               in uint32 current_value,
                               in uint32 max_value,
                               in uint32 queuing_disc,
                               out uint32 semaphore_id,
                               out uint32 return_code);

    signature DISPLAY_BLACKBOARD(in uint32 blackboard_id,
                                 in uint32 msg_addr,
                                 in uint32 msg_length,
                                 out uint32 return_code);

    signature READ_SAMPLING_MESSAGE(in uint32 port_id,
                                    in uint32 msg_addr,
                                    out uint32 msg_length,
                                    out uint32 validity,
                                    out uint32 return_code);

    signature RESET_EVENT(in uint32 event_id,
                          out uint32 return_code);

    signature SET_EVENT(in uint32 event_id,
                        out uint32 return_code);

    signature SIGNAL_SEMAPHORE(in uint32 semaphore_id,
                               out uint32 return_code);

    signature START(in uint32 process_id,
                out uint32 return_code);

signature STOP(in uint32 process_id,
               out uint32 return_code);

signature STOP_SELF();

signature WRITE_SAMPLING_MESSAGE(in uint32 port_id,
                                 in uint32 msg_addr,
                                 in uint32 msg_length,
                                 out uint32 return_code);

signature READ_BLACKBOARD(in uint32 blackboard_id,
                          in uint32 time_out,
                          in uint32 msg_addr,
                          out uint32 msg_length,
                          out uint32 return_code);

signature RECEIVE_BUFFER(in uint32 buffer_id,
                         in uint32 time_out,
                         in uint32 msg_addr,
                         out uint32 msg_length,
                         out uint32 return_code);

signature RECEIVE_QUEUING_MESSAGE(in uint32 port_id,
                                  in uint32 time_out,
                                  in uint32 msg_addr,
                                  out uint32 msg_length,
                                  out uint32 return_code);

signature SEND_BUFFER(in uint32 buffer_id,
                      in uint32 msg_addr,
                      in uint32 msg_length,
                      in uint32 time_out,
                      out uint32 return_code);

signature SEND_QUEUING_MESSAGE(in uint32 port_id,
                               in uint32 msg_addr,
                               in uint32 msg_length,
                               in uint32 time_out,
                               out uint32 return_code);

signature WAIT_EVENT(in uint32 event_id,
                     in uint32 time_out,
                     out uint32 return_code);

signature WAIT_SEMAPHORE(in uint32 semaphore_id,
                         in uint32 time_out,
                         out uint32 return_code);

signature SET_PARTITION_MODE(in uint32 operating_mode,
                             out uint32 return_code);

signature GET_PARTITION_STATUS(out uint32 id,
                               out uint32 period,
                               out uint32 duration,
                               out uint32 lock_level,
                               out uint32 operating_mode,
                               out uint32 start_condition,
                               out uint32 return_code);

signature GET_PROCESS_ID(in uint32 process_name,
                         out uint32 process_id,
                         out uint32 return_code);

// auxiliary functions
signature GET_DATA_TABLE_ENTRY(in uint32 idx,
                              out uint32 return_code,
                              out charstring data);

signature SET_DATA_TABLE_ENTRY(in uint32 idx,
                              in charstring data,
                              out uint32 return_code);

signature CLEAR_DATA_TABLE_ENTRY(in uint32 idx,
                                out uint32 return_code);

// scenario calls
signature OPEN_SIGNAL_PORTS(out uint32 succeeded,
                           out uint32 failed,
                           out uint32 return_code);

signature READ_SIGNAL(in uint32 idx,
                     out uint32 signal_value,
                     out uint32 fs,
                     out uint32 return_code);

signature WRITE_SIGNAL(in uint32 idx,
                      in uint32 signal_value,
                      in uint32 fs,
                      out uint32 return_code);

signature STEPPER_INIT(in uint32 idx,
                      out uint32 return_code);

signature STEPPER_SET(in uint32 idx,
                     in uint32 cmd,
                     in uint32 angle,
                     out uint32 return_code);

signature STEPPER_GET(in uint32 idx,
                     out uint32 status,
                     out uint32 angle,
                     out uint32 return_code);

type port CmdPort procedure {
    inout set_process_idx, CLEAR_BLACKBOARD, CREATE_BLACKBOARD, CREATE_BUFFER,
          CREATE_EVENT, CREATE_PROCESS, CREATE_QUEUING_PORT, ↩
              ↪ CREATE_SAMPLING_PORT,
          CREATE_SEMAPHORE, DISPLAY_BLACKBOARD, READ_SAMPLING_MESSAGE,
          RESET_EVENT, SET_EVENT, SIGNAL_SEMAPHORE, START, STOP, STOP_SELF,
          WRITE_SAMPLING_MESSAGE, READ_BLACKBOARD, RECEIVE_BUFFER,
          RECEIVE_QUEUING_MESSAGE, SEND_BUFFER, SEND_QUEUING_MESSAGE, ↩
              ↪ WAIT_EVENT,
          WAIT_SEMAPHORE, SET_PARTITION_MODE,
          GET_PARTITION_STATUS, GET_PROCESS_ID,
          GET_DATA_TABLE_ENTRY, SET_DATA_TABLE_ENTRY, CLEAR_DATA_TABLE_ENTRY,
          OPEN_SIGNAL_PORTS, READ_SIGNAL, WRITE_SIGNAL,
          STEPPER_INIT, STEPPER_SET, STEPPER_GET;
};

type port AFDX_In_Port message {
    in octetstring
};

type port AFDX_Out_Port message {
    out octetstring
};

type port TACP_In_Port message {
    in tacp_l1_pdu
};

type port TACP_Out_Port message {
        out tacp_l1_pdu
    };

    // float helper functions
    function intAsFloat(in uint32 i) return float32 {
        var float32 f;
        var bitstring bs := int2bit(i, 32);
        var integer r := decvalue(bs, f);
        return f;
    }

    function floatAsInt(in float32 f) return uint32 {
        var bitstring bs := encvalue(f);
        var uint32 i := bit2int(bs);
        return i;
    }

    function abs(in float32 f) return float32 {
        if (f < 0.0) {
            return -f;
        }
        else {
            return f;
        }
    }
}
```

C.6 TACP Protocol Handler

```
module TACP {

    import from types all;

    template tacp_l1_pdu l1_any := ?;
    template tacp_l1_pdu l1_ack := {
        seq_no := ?,
        ack_no := ?,
        flags := 0,
        spare := 0,
        l2 := omit
    };
    template tacp_l1_pdu l1_initial := {
        seq_no := 0,
        ack_no := 0,
        flags := 0,
        spare := 0,
        l2 := omit
    };
    template tacp_l1_pdu l1_apicall := {
        seq_no := 0,
        ack_no := 0,
        flags := 0,
        spare := 0,
        l2 := {
            partition := 9, // not used here
            process_idx := 0,
            operation := { INVALID_OPERATION_cmd := {} }
        }
    };

    const integer rbLength := 8;
    type record length (rbLength) of tacp_l1_pdu Ringbuffer;
    type integer RingbufferIdx (0..rbLength-1);
    type record SendQueue {
        Ringbuffer rb,
    RingbufferIdx ri,
    RingbufferIdx wi
};

function queuePut(inout SendQueue q, in tacp_l1_pdu data) return boolean {
    var integer nwi := (q.wi + 1) mod rbLength;
    if (nwi != q.ri) {
        q.rb[q.wi] := data;
        q.wi := nwi;
        return true;
    }
    else {
        return false;
    }
}

function queueGet(inout SendQueue q, out tacp_l1_pdu data) return boolean {
    if (q.ri != q.wi) {
        data := q.rb[q.ri];
        q.ri := (q.ri + 1) mod rbLength;
        return true;
    }
    else {
        return false;
    }
}

function queueEmpty(inout SendQueue q) return boolean {
    return q.ri == q.wi;
}

type component TACPHandler {
    var tacp_l1_pdu last_l1_pdu := l1_initial;
    var uint8 last_recv_seq := 0;
    var uint8 last_recv_ack := 0;
    var uint8 last_sent_seq := 0;
    var uint8 last_sent_ack := 0;
    var uint8 process_idx := 0;
    var boolean firstPacket := true;
    var SendQueue sendq;

    port CmdPort cmd;
    port TACP_Out_Port tacpToCPM;
    port TACP_In_Port tacpFromCPM;
};

altstep apicall() runs on TACPHandler {
    var tacp_l1_pdu l1_pdu := l1_apicall;
    var Operation op;

    [] cmd.getcall(set_process_idx:{?}) ->
        param(process_idx) {
        cmd.reply(set_process_idx:{-});
    }

    [] cmd.getcall(CLEAR_BLACKBOARD:{?,-}) ->
        param(op.API_CLEAR_BLACKBOARD_cmd.blackboard_id) {
        l1_pdu.l2.process_idx := process_idx;
        l1_pdu.l2.operation := op;
            queuePut(sendq, l1_pdu);
    }

    [] cmd.getcall(CREATE_BLACKBOARD:{?,?,-,-}) ->
        param(op.API_CREATE_BLACKBOARD_cmd.name,
              op.API_CREATE_BLACKBOARD_cmd.max_msg_size) {
        l1_pdu.l2.process_idx := process_idx;
        l1_pdu.l2.operation := op;
        queuePut(sendq, l1_pdu);
}

[] cmd.getcall(CREATE_BUFFER:{?,?,?,?,-,-}) ->
    param(op.API_CREATE_BUFFER_cmd.name,
          op.API_CREATE_BUFFER_cmd.max_msg_size,
          op.API_CREATE_BUFFER_cmd.max_nb_msg,
          op.API_CREATE_BUFFER_cmd.queuing_disc) {
    l1_pdu.l2.process_idx := process_idx;
    l1_pdu.l2.operation := op;
    queuePut(sendq, l1_pdu);
}

[] cmd.getcall(CREATE_EVENT:{?,-,-}) ->
    param(op.API_CREATE_EVENT_cmd.name) {
    l1_pdu.l2.process_idx := process_idx;
    l1_pdu.l2.operation := op;
    queuePut(sendq, l1_pdu);
}

[] cmd.getcall(CREATE_PROCESS:{?,?,?,?,?,?,?,-,-}) ->
        param(op.API_CREATE_PROCESS_cmd.name,
              op.API_CREATE_PROCESS_cmd.entry_point,
              op.API_CREATE_PROCESS_cmd.stack_size,
              op.API_CREATE_PROCESS_cmd.base_priority,
              op.API_CREATE_PROCESS_cmd.period,
              op.API_CREATE_PROCESS_cmd.time_capacity,
              op.API_CREATE_PROCESS_cmd.deadline) {
    l1_pdu.l2.process_idx := process_idx;
    l1_pdu.l2.operation := op;
        queuePut(sendq, l1_pdu);
}

[] cmd.getcall(CREATE_QUEUING_PORT:{?,?,?,?,?,-,-}) ->
        param(op.API_CREATE_QUEUING_PORT_cmd.name,
              op.API_CREATE_QUEUING_PORT_cmd.max_msg_size,
              op.API_CREATE_QUEUING_PORT_cmd.max_nb_msg,
              op.API_CREATE_QUEUING_PORT_cmd.port_direction,
              op.API_CREATE_QUEUING_PORT_cmd.queuing_disc) {
    l1_pdu.l2.process_idx := process_idx;
    l1_pdu.l2.operation := op;
    queuePut(sendq, l1_pdu);
}

[] cmd.getcall(CREATE_SAMPLING_PORT:{?,?,?,?,-,-}) ->
        param(op.API_CREATE_SAMPLING_PORT_cmd.name,
              op.API_CREATE_SAMPLING_PORT_cmd.max_msg_size,
              op.API_CREATE_SAMPLING_PORT_cmd.port_direction,
              op.API_CREATE_SAMPLING_PORT_cmd.refresh_period) {
    l1_pdu.l2.process_idx := process_idx;
    l1_pdu.l2.operation := op;
    queuePut(sendq, l1_pdu);
}

[] cmd.getcall(CREATE_SEMAPHORE:{?,?,?,?,-,-}) ->
        param(op.API_CREATE_SEMAPHORE_cmd.name,
              op.API_CREATE_SEMAPHORE_cmd.current_value,
              op.API_CREATE_SEMAPHORE_cmd.max_value,
              op.API_CREATE_SEMAPHORE_cmd.queuing_disc) {
    l1_pdu.l2.process_idx := process_idx;
    l1_pdu.l2.operation := op;
    queuePut(sendq, l1_pdu);
}

[] cmd.getcall(DISPLAY_BLACKBOARD:{?,?,?,-}) ->
        param(op.API_DISPLAY_BLACKBOARD_cmd.blackboard_id,
              op.API_DISPLAY_BLACKBOARD_cmd.msg_addr,
                    op.API_DISPLAY_BLACKBOARD_cmd.msg_length) {
        l1_pdu.l2.process_idx := process_idx;
        l1_pdu.l2.operation := op;
        queuePut(sendq, l1_pdu);
    }

    [] cmd.getcall(READ_SAMPLING_MESSAGE:{?,?,-,-,-}) ->
            param(op.API_READ_SAMPLING_MESSAGE_cmd.port_id,
                  op.API_READ_SAMPLING_MESSAGE_cmd.msg_addr) {
        l1_pdu.l2.process_idx := process_idx;
        l1_pdu.l2.operation := op;
        queuePut(sendq, l1_pdu);
    }

    [] cmd.getcall(RESET_EVENT:{?,-}) ->
        param(op.API_RESET_EVENT_cmd.event_id) {
        l1_pdu.l2.process_idx := process_idx;
        l1_pdu.l2.operation := op;
        queuePut(sendq, l1_pdu);
    }

    [] cmd.getcall(SET_EVENT:{?,-}) ->
        param(op.API_SET_EVENT_cmd.event_id) {
        l1_pdu.l2.process_idx := process_idx;
        l1_pdu.l2.operation := op;
        queuePut(sendq, l1_pdu);
    }

    [] cmd.getcall(SIGNAL_SEMAPHORE:{?,-}) ->
        param(op.API_SIGNAL_SEMAPHORE_cmd.semaphore_id) {
        l1_pdu.l2.process_idx := process_idx;
        l1_pdu.l2.operation := op;
        queuePut(sendq, l1_pdu);
    }

    [] cmd.getcall(START:{?,-}) ->
        param(op.API_START_cmd.process_id) {
        l1_pdu.l2.process_idx := process_idx;
        l1_pdu.l2.operation := op;
        queuePut(sendq, l1_pdu);
    }

    [] cmd.getcall(STOP:{?,-}) ->
        param(op.API_STOP_cmd.process_id) {
        l1_pdu.l2.process_idx := process_idx;
        l1_pdu.l2.operation := op;
        queuePut(sendq, l1_pdu);
    }

    [] cmd.getcall(STOP_SELF:{}) {
        l1_pdu.l2.process_idx := process_idx;
        l1_pdu.l2.operation.API_STOP_SELF_cmd := {};
        queuePut(sendq, l1_pdu);
    }

    [] cmd.getcall(WRITE_SAMPLING_MESSAGE:{?,?,?,-}) ->
            param(op.API_WRITE_SAMPLING_MESSAGE_cmd.port_id,
                  op.API_WRITE_SAMPLING_MESSAGE_cmd.msg_addr,
                  op.API_WRITE_SAMPLING_MESSAGE_cmd.msg_length) {
        l1_pdu.l2.process_idx := process_idx;
        l1_pdu.l2.operation := op;
        queuePut(sendq, l1_pdu);
    }

    [] cmd.getcall(READ_BLACKBOARD:{?,?,?,-,-}) ->
            param(op.API_READ_BLACKBOARD_cmd.blackboard_id,
                  op.API_READ_BLACKBOARD_cmd.time_out,
              op.API_READ_BLACKBOARD_cmd.msg_addr) {
    l1_pdu.l2.process_idx := process_idx;
    l1_pdu.l2.operation := op;
    queuePut(sendq, l1_pdu);
}

[] cmd.getcall(RECEIVE_BUFFER:{?,?,?,-,-}) ->
        param(op.API_RECEIVE_BUFFER_cmd.buffer_id,
              op.API_RECEIVE_BUFFER_cmd.time_out,
              op.API_RECEIVE_BUFFER_cmd.msg_addr) {
    l1_pdu.l2.process_idx := process_idx;
    l1_pdu.l2.operation := op;
    queuePut(sendq, l1_pdu);
}

[] cmd.getcall(RECEIVE_QUEUING_MESSAGE:{?,?,?,-,-}) ->
        param(op.API_RECEIVE_QUEUING_MESSAGE_cmd.port_id,
              op.API_RECEIVE_QUEUING_MESSAGE_cmd.time_out,
              op.API_RECEIVE_QUEUING_MESSAGE_cmd.msg_addr) {
    l1_pdu.l2.process_idx := process_idx;
    l1_pdu.l2.operation := op;
    queuePut(sendq, l1_pdu);
}

[] cmd.getcall(SEND_BUFFER:{?,?,?,?,-}) ->
        param(op.API_SEND_BUFFER_cmd.buffer_id,
              op.API_SEND_BUFFER_cmd.msg_addr,
              op.API_SEND_BUFFER_cmd.msg_length,
              op.API_SEND_BUFFER_cmd.time_out) {
    l1_pdu.l2.process_idx := process_idx;
    l1_pdu.l2.operation := op;
    queuePut(sendq, l1_pdu);
}

[] cmd.getcall(SEND_QUEUING_MESSAGE:{?,?,?,?,-}) ->
        param(op.API_SEND_QUEUING_MESSAGE_cmd.port_id,
              op.API_SEND_QUEUING_MESSAGE_cmd.msg_addr,
              op.API_SEND_QUEUING_MESSAGE_cmd.msg_length,
              op.API_SEND_QUEUING_MESSAGE_cmd.time_out) {
    l1_pdu.l2.process_idx := process_idx;
    l1_pdu.l2.operation := op;
    queuePut(sendq, l1_pdu);
}

[] cmd.getcall(WAIT_EVENT:{?,?,-}) ->
    param(op.API_WAIT_EVENT_cmd.event_id,
          op.API_WAIT_EVENT_cmd.time_out) {
    l1_pdu.l2.process_idx := process_idx;
    l1_pdu.l2.operation := op;
    queuePut(sendq, l1_pdu);
}

[] cmd.getcall(WAIT_SEMAPHORE:{?,?,-}) ->
    param(op.API_WAIT_SEMAPHORE_cmd.semaphore_id,
          op.API_WAIT_SEMAPHORE_cmd.time_out) {
    l1_pdu.l2.process_idx := process_idx;
    l1_pdu.l2.operation := op;
    queuePut(sendq, l1_pdu);
}

[] cmd.getcall(SET_PARTITION_MODE:{?,-}) ->
    param(op.API_SET_PARTITION_MODE_cmd.operating_mode) {
    l1_pdu.l2.process_idx := process_idx;
    l1_pdu.l2.operation := op;
    queuePut(sendq, l1_pdu);
}
```

```
[] cmd.getcall(GET_PARTITION_STATUS:{-,-,-,-,-,-,-}) {
    l1_pdu.l2.process_idx := process_idx;
    l1_pdu.l2.operation.API_GET_PARTITION_STATUS_cmd := {};
    queuePut(sendq, l1_pdu);
}

[] cmd.getcall(GET_PROCESS_ID:{?,-,-}) ->
    param(op.API_GET_PROCESS_ID_cmd.name) {
    l1_pdu.l2.process_idx := process_idx;
    l1_pdu.l2.operation := op;
    queuePut(sendq, l1_pdu);
}

[] cmd.getcall(GET_DATA_TABLE_ENTRY:{?,-,-}) ->
    param(op.AUX_GET_DATA_TABLE_ENTRY_cmd.idx) {
    l1_pdu.l2.process_idx := process_idx;
    l1_pdu.l2.operation := op;
    queuePut(sendq, l1_pdu);
}

[] cmd.getcall(SET_DATA_TABLE_ENTRY:{?,?,-}) ->
    param(op.AUX_SET_DATA_TABLE_ENTRY_cmd.idx,
          op.AUX_SET_DATA_TABLE_ENTRY_cmd.data) {
    l1_pdu.l2.process_idx := process_idx;
    l1_pdu.l2.operation := op;
    queuePut(sendq, l1_pdu);
}

[] cmd.getcall(CLEAR_DATA_TABLE_ENTRY:{?,-}) ->
    param(op.AUX_CLEAR_DATA_TABLE_ENTRY_cmd.idx) {
    l1_pdu.l2.process_idx := process_idx;
    l1_pdu.l2.operation := op;
    queuePut(sendq, l1_pdu);
}

[] cmd.getcall(OPEN_SIGNAL_PORTS:{-,-,-}) {
    l1_pdu.l2.process_idx := process_idx;
    l1_pdu.l2.operation.SCE_OPEN_SIGNAL_PORTS_cmd := {};
    queuePut(sendq, l1_pdu);
}

[] cmd.getcall(READ_SIGNAL:{?,-,-,-}) ->
    param(op.SCE_READ_SIGNAL_cmd.idx) {
    l1_pdu.l2.process_idx := process_idx;
    l1_pdu.l2.operation := op;
    queuePut(sendq, l1_pdu);
}

[] cmd.getcall(WRITE_SIGNAL:{?,?,?,-}) ->
        param(op.SCE_WRITE_SIGNAL_cmd.idx,
              op.SCE_WRITE_SIGNAL_cmd.signal_value,
              op.SCE_WRITE_SIGNAL_cmd.fs) {
    l1_pdu.l2.process_idx := process_idx;
    l1_pdu.l2.operation := op;
    queuePut(sendq, l1_pdu);
}

[] cmd.getcall(STEPPER_INIT:{?,-}) ->
    param(op.SCE_STEPPER_INIT_cmd.idx) {
    l1_pdu.l2.process_idx := process_idx;
    l1_pdu.l2.operation := op;
    queuePut(sendq, l1_pdu);
}

[] cmd.getcall(STEPPER_SET:{?,?,?,-}) ->
        param(op.SCE_STEPPER_SET_cmd.idx,
              op.SCE_STEPPER_SET_cmd.cmd,
                op.SCE_STEPPER_SET_cmd.angle) {
        l1_pdu.l2.process_idx := process_idx;
        l1_pdu.l2.operation := op;
        queuePut(sendq, l1_pdu);
    }

    [] cmd.getcall(STEPPER_GET:{?,-,-,-}) ->
        param(op.SCE_STEPPER_GET_cmd.idx) {
        l1_pdu.l2.process_idx := process_idx;
        l1_pdu.l2.operation := op;
        queuePut(sendq, l1_pdu);
    }
}


altstep response() runs on TACPHandler {
    var tacp_l1_pdu l1_pdu;

    [] tacpFromCPM.receive(l1_any) -> value l1_pdu {
        last_recv_ack := l1_pdu.ack_no;
        if (firstPacket) {
            firstPacket := false;
            last_sent_ack := l1_pdu.seq_no;
            last_recv_seq := l1_pdu.seq_no;
            last_sent_seq := l1_pdu.ack_no;
            last_l1_pdu.seq_no := last_recv_ack;
            last_l1_pdu.seq_no := last_recv_ack;
            last_l1_pdu.seq_no := last_recv_ack;
            last_l1_pdu.ack_no := last_recv_seq;
        }
        if (last_recv_seq != l1_pdu.seq_no) {
            if (ispresent(l1_pdu.l2) and ↩
                 ↪ ischosen(l1_pdu.l2.operation.INVALID_OPERATION_resp)) {
                log(”skipped invalid TACP packet”);
                goto skip;
            }
            last_recv_seq := l1_pdu.seq_no;
            log(”received new valid TACP response”);

            if (ispresent(l1_pdu.l2)) {
                log(”layer 2 present”);
                var Operation op := l1_pdu.l2.operation;

                if (ischosen(op.API_CLEAR_BLACKBOARD_resp)) {
                    cmd.reply(CLEAR_BLACKBOARD:
                        {-,op.API_CLEAR_BLACKBOARD_resp.return_code});
                }
                else if (ischosen(op.API_CREATE_BLACKBOARD_resp)) {
                    cmd.reply(CREATE_BLACKBOARD:
                        {-,-,
                         op.API_CREATE_BLACKBOARD_resp.blackboard_id,
                         op.API_CREATE_BLACKBOARD_resp.return_code});
                }
                else if (ischosen(op.API_CREATE_BUFFER_resp)) {
                    cmd.reply(CREATE_BUFFER:
                        {-,-,-,-,
                         op.API_CREATE_BUFFER_resp.buffer_id,
                         op.API_CREATE_BUFFER_resp.return_code});
                }
                else if (ischosen(op.API_CREATE_EVENT_resp)) {
                    cmd.reply(CREATE_EVENT:
                        {-,
                         op.API_CREATE_EVENT_resp.event_id,
                         op.API_CREATE_EVENT_resp.return_code});
                }
                else if (ischosen(op.API_CREATE_PROCESS_resp)) {
                    cmd.reply(CREATE_PROCESS:
```

```
        {-,-,-,-,-,-,-,
         op.API_CREATE_PROCESS_resp.process_id,
         op.API_CREATE_PROCESS_resp.return_code});
}
else if (ischosen(op.API_CREATE_QUEUING_PORT_resp)) {
    cmd.reply(CREATE_QUEUING_PORT:
        {-,-,-,-,-,
         op.API_CREATE_QUEUING_PORT_resp.port_id,
         op.API_CREATE_QUEUING_PORT_resp.return_code});
}
else if (ischosen(op.API_CREATE_SAMPLING_PORT_resp)) {
    cmd.reply(CREATE_SAMPLING_PORT:
        {-,-,-,-,
         op.API_CREATE_SAMPLING_PORT_resp.port_id,
         op.API_CREATE_SAMPLING_PORT_resp.return_code});
}
else if (ischosen(op.API_CREATE_SEMAPHORE_resp)) {
    cmd.reply(CREATE_SEMAPHORE:
        {-,-,-,-,
         op.API_CREATE_SEMAPHORE_resp.semaphore_id,
         op.API_CREATE_SEMAPHORE_resp.return_code});
}
else if (ischosen(op.API_DISPLAY_BLACKBOARD_resp)) {
    cmd.reply(DISPLAY_BLACKBOARD:
        {-,-,-,
         op.API_DISPLAY_BLACKBOARD_resp.return_code});
}
else if (ischosen(op.API_READ_SAMPLING_MESSAGE_resp)) {
    cmd.reply(READ_SAMPLING_MESSAGE:
        {-,-,
         op.API_READ_SAMPLING_MESSAGE_resp.msg_length,
         op.API_READ_SAMPLING_MESSAGE_resp.validity,
         op.API_READ_SAMPLING_MESSAGE_resp.return_code});
}
else if (ischosen(op.API_RESET_EVENT_resp)) {
    cmd.reply(RESET_EVENT:
        {-,
         op.API_RESET_EVENT_resp.return_code});
}
else if (ischosen(op.API_SET_EVENT_resp)) {
    cmd.reply(SET_EVENT:
        {-,
         op.API_SET_EVENT_resp.return_code});
}
else if (ischosen(op.API_SIGNAL_SEMAPHORE_resp)) {
    cmd.reply(SIGNAL_SEMAPHORE:
        {-,
         op.API_SIGNAL_SEMAPHORE_resp.return_code});
}
else if (ischosen(op.API_START_resp)) {
    cmd.reply(START:
        {-,
         op.API_START_resp.return_code});
}
else if (ischosen(op.API_STOP_resp)) {
    cmd.reply(STOP:
        {-,
         op.API_STOP_resp.return_code});
}
else if (ischosen(op.API_STOP_SELF_resp)) {
    cmd.reply(STOP_SELF:{});
}
else if (ischosen(op.API_WRITE_SAMPLING_MESSAGE_resp)) {
    cmd.reply(WRITE_SAMPLING_MESSAGE:
        {-,-,-,
         op.API_WRITE_SAMPLING_MESSAGE_resp.return_code});
}
```

```
else if (ischosen(op.API_READ_BLACKBOARD_resp)) {
    cmd.reply(READ_BLACKBOARD:
        {-,-,-,
         op.API_READ_BLACKBOARD_resp.msg_length,
         op.API_READ_BLACKBOARD_resp.return_code});
}
else if (ischosen(op.API_RECEIVE_BUFFER_resp)) {
    cmd.reply(RECEIVE_BUFFER:
        {-,-,-,
         op.API_RECEIVE_BUFFER_resp.msg_length,
         op.API_RECEIVE_BUFFER_resp.return_code});
}
else if (ischosen(op.API_RECEIVE_QUEUING_MESSAGE_resp)) {
    cmd.reply(RECEIVE_QUEUING_MESSAGE:
        {-,-,-,
         op.API_RECEIVE_QUEUING_MESSAGE_resp.msg_length,
         op.API_RECEIVE_QUEUING_MESSAGE_resp.return_code});
}
else if (ischosen(op.API_SEND_BUFFER_resp)) {
    cmd.reply(SEND_BUFFER:
        {-,-,-,-,
         op.API_SEND_BUFFER_resp.return_code});
}
else if (ischosen(op.API_SEND_QUEUING_MESSAGE_resp)) {
    cmd.reply(SEND_QUEUING_MESSAGE:
        {-,-,-,-,
         op.API_SEND_QUEUING_MESSAGE_resp.return_code});
}
else if (ischosen(op.API_WAIT_EVENT_resp)) {
    cmd.reply(WAIT_EVENT:
        {-,-,
         op.API_WAIT_EVENT_resp.return_code});
}
else if (ischosen(op.API_WAIT_SEMAPHORE_resp)) {
    cmd.reply(WAIT_SEMAPHORE:
        {-,-,
         op.API_WAIT_SEMAPHORE_resp.return_code});
}
else if (ischosen(op.API_SET_PARTITION_MODE_resp)) {
    cmd.reply(SET_PARTITION_MODE:
        {-,
         op.API_SET_PARTITION_MODE_resp.return_code});
}
else if (ischosen(op.API_GET_PARTITION_STATUS_resp)) {
    cmd.reply(GET_PARTITION_STATUS:
        {op.API_GET_PARTITION_STATUS_resp.id,
         op.API_GET_PARTITION_STATUS_resp.period,
         op.API_GET_PARTITION_STATUS_resp.duration,
         op.API_GET_PARTITION_STATUS_resp.lock_level,
         op.API_GET_PARTITION_STATUS_resp.operating_mode,
         op.API_GET_PARTITION_STATUS_resp.start_condition,
         op.API_GET_PARTITION_STATUS_resp.return_code});
}
else if (ischosen(op.API_GET_PROCESS_ID_resp)) {
    cmd.reply(GET_PROCESS_ID:
        {-,
         op.API_GET_PROCESS_ID_resp.process_id,
         op.API_GET_PROCESS_ID_resp.return_code});
}
else if (ischosen(op.AUX_GET_DATA_TABLE_ENTRY_resp)) {
    cmd.reply(GET_DATA_TABLE_ENTRY:
        {-,
         op.AUX_GET_DATA_TABLE_ENTRY_resp.return_code,
         op.AUX_GET_DATA_TABLE_ENTRY_resp.data});
}
else if (ischosen(op.AUX_SET_DATA_TABLE_ENTRY_resp)) {
    cmd.reply(SET_DATA_TABLE_ENTRY:
```

```
                            {-,-,
                             op.AUX_GET_DATA_TABLE_ENTRY_resp.return_code});
                    }
                    else if (ischosen(op.AUX_CLEAR_DATA_TABLE_ENTRY_resp)) {
                        cmd.reply(CLEAR_DATA_TABLE_ENTRY:
                            {-,
                             op.AUX_CLEAR_DATA_TABLE_ENTRY_resp.return_code});
                    }
                    else if (ischosen(op.SCE_OPEN_SIGNAL_PORTS_resp)) {
                        cmd.reply(OPEN_SIGNAL_PORTS:
                            {op.SCE_OPEN_SIGNAL_PORTS_resp.succeeded,
                             op.SCE_OPEN_SIGNAL_PORTS_resp.failed,
                             op.SCE_OPEN_SIGNAL_PORTS_resp.return_code});
                    }
                    else if (ischosen(op.SCE_READ_SIGNAL_resp)) {
                        cmd.reply(READ_SIGNAL:
                            {-,
                             op.SCE_READ_SIGNAL_resp.signal_value,
                             op.SCE_READ_SIGNAL_resp.fs,
                             op.SCE_READ_SIGNAL_resp.return_code});
                    }
                    else if (ischosen(op.SCE_WRITE_SIGNAL_resp)) {
                        cmd.reply(WRITE_SIGNAL:
                            {-,-,-,
                             op.SCE_WRITE_SIGNAL_resp.return_code});
                    }
                    else if (ischosen(op.SCE_STEPPER_INIT_resp)) {
                        cmd.reply(STEPPER_INIT:
                            {-,
                             op.SCE_STEPPER_INIT_resp.return_code});
                    }
                    else if (ischosen(op.SCE_STEPPER_SET_resp)) {
                        cmd.reply(STEPPER_SET:
                            {-,-,-,
                             op.SCE_STEPPER_SET_resp.return_code});
                    }
                    else if (ischosen(op.SCE_STEPPER_GET_resp)) {
                        cmd.reply(STEPPER_GET:
                            {-,
                             op.SCE_STEPPER_GET_resp.status,
                             op.SCE_STEPPER_GET_resp.angle,
                             op.SCE_STEPPER_GET_resp.return_code});
                    }
                }
            }
            label skip;
        }
    }


    function runHandler() runs on TACPHandler {

        sendq.ri := 0;
        sendq.wi := 0;

        while (true) {
            alt {
                [] apicall();
                [] response();
            }

            if (last_recv_ack == last_sent_seq and not queueEmpty(sendq)) {
                queueGet(sendq, last_l1_pdu);
                last_sent_seq := (last_sent_seq + 1) mod 256;
                last_l1_pdu.seq_no := last_sent_seq;
                last_sent_ack := last_recv_seq;
                last_l1_pdu.ack_no := last_sent_ack;
            }
            else if (last_recv_seq != last_sent_ack) {
                last_sent_ack := last_recv_seq;
                last_l1_pdu.ack_no := last_sent_ack;
            }
            tacpToCPM.send(last_l1_pdu);
        }
    }
}
```

APPENDIX D

Model Validity Checkers

The generators presented in this section are written in MERL (cf. section 3.2.4). They do not generate regular outputs, but instead they check the models built with MetaEdit+ for conformance to additional constraints that are not directly supported in MetaEdit+.

D.1 Configuration Validator

```
report '!Validate Configuration'

/* throw away all output */
to '%null
* $'
endto

/* The partition slice duration must be less than
   the partition scheduling period. */
foreach .Partition
{
  if (:Duration >= :Period; num) then
    'Error: ' type ': The slice duration must be '
    'less than the partition period.' newline
  endif
}

/* An AFDX Port Group must be in exactly one Dataflow
   relationship. */
foreach .AFDX Port Group;
{
  $count = '0'
  do >Dataflow
  {
    $count++%null
  }
  if ($count <> '1' num) then
    'Error: ' type ': Must be connected '
    'to exactly one partition.' newline
  endif
}

/* A RAM Port Group must be in exactly 1 Sender role. */
foreach .RAM Port Group;
{
  $count = '0'
  do ~Sender
  {
    $count++%null
  }
  if ($count <> '1' num) then
    'Error: ' type ': Must be in exactly '
    'one Sender role.' newline
  endif
}

/* A RAM Port Group must be in exactly 1 Receiver role. */
foreach .RAM Port Group;
{
  $count = '0'
  do ~Receiver
  {
    $count++%null
  }
  if ($count <> '1' num) then
    'Error: ' type ': Must be in exactly '
    'one Receiver role.' newline
  endif
}

/* A Partition must be in exactly 1 Containee role. */
foreach .Partition
{
  $count = '0'
  do ~Containee
  {
    $count++%null
  }
  if ($count <> '1' num) then
    'Error: ' type ': Must be connected to exactly '
    'one CPM.' newline
  endif
}

endreport
```

D.2 Behaviour Validator

```
report '!Validate Behaviour'

/* throw away all output */
to '%null
* $'
endto

/* make into a space-separated wildcard '* xxx *' */
to '%wildsp
/^(.*)$/ $\*\ $1\ \*'
endto

/* Test Start must occur exactly 1 time */
$count = '0'
foreach .Test Start
{
  $count++%null
}
if ($count <> ’1’ num) then
  ’Error: Test Start: Must occur exactly one time.’ newline
endif

/* Clear_Blackboard must be in exactly 1 API call [Blackboard] role */
/* Create_Blackboard must be in exactly 1 API call [Blackboard] role */
/* Display_Blackboard must be in exactly 1 API call [Blackboard] role */
/* Read_Blackboard must be in exactly 1 API call [Blackboard] role */
foreach .(Clear_Blackboard | Create_Blackboard | Display_Blackboard |
          Read_Blackboard)
{
  $count = ’0’
  do ~API call [Blackboard]
  {
    $count++%null
  }
  if ($count <> ’1’ num) then
    ’Error: ’ type ’: Must be connected to exactly one ’
    ’Blackboard.’ newline
  endif
}

/* Create_Buffer must be in exactly 1 API call [Buffer] role */
/* Receive_Buffer must be in exactly 1 API call [Buffer] role */
/* Send_Buffer must be in exactly 1 API call [Buffer] role */
foreach .(Create_Buffer | Receive_Buffer | Send_Buffer)
{
  $count = ’0’
  do ~API call [Buffer]
  {
    $count++%null
  }
  if ($count <> ’1’ num) then
    ’Error: ’ type ’: Must be connected to exactly one ’
    ’Buffer.’ newline
  endif
}

/* Create_Event must be in exactly 1 API call [Event] role */
/* Reset_Event must be in exactly 1 API call [Event] role */
/* Set_Event must be in exactly 1 API call [Event] role */
/* Wait_Event must be in exactly 1 API call [Event] role */
foreach .(Create_Event | Reset_Event | Set_Event | Wait_Event)
{
  $count = ’0’
  do ~API call [Event]
  {
    $count++%null
  }
  if ($count <> ’1’ num) then
    ’Error: ’ type ’: Must be connected to exactly one ’
    ’Event.’ newline
  endif
}

/* Create_Semaphore must be in exactly 1 API call [Semaphore] role */
/* Signal_Semaphore must be in exactly 1 API call [Semaphore] role */
/* Wait_Semaphore must be in exactly 1 API call [Semaphore] role */
foreach .(Create_Semaphore | Signal_Semaphore | Wait_Semaphore)
{
  $count = ’0’
  do ~API call [Semaphore]
  {
    $count++%null
  }
  if ($count <> '1' num) then
    'Error: ' type ': Must be connected to exactly one '
    'Semaphore.' newline
  endif
}

/* Create_Process must be in exactly 1 API call [Process] role */
/* Start must be in exactly 1 API call [Process] role */
/* Stop must be in exactly 1 API call [Process] role */
foreach .(Create_Process | Start | Stop)
{
  $count = '0'
  do ~API call [Process]
  {
    $count++%null
  }
  if ($count <> '1' num) then
    'Error: ' type ': Must be connected to exactly one '
    'Process.' newline
  endif
}

/* Create_Queuing_Port must be in exactly 1 API call [Port] role */
/* Create_Sampling_Port must be in exactly 1 API call [Port] role */
/* Read_Sampling_Message must be in exactly 1 API call [Port] role */
/* Receive_Queuing_Message must be in exactly 1 API call [Port] role */
/* Send_Queuing_Message must be in exactly 1 API call [Port] role */
/* Write_Sampling_Message must be in exactly 1 API call [Port] role */
/* Complement must be in exactly 1 API call [Port] role */
foreach .(Create_Queuing_Port | Create_Sampling_Port |
          Read_Sampling_Message | Receive_Queuing_Message |
          Send_Queuing_Message | Write_Sampling_Message | Complement)
{
  $count = '0'
  do ~API call [Port]
  {
    $count++%null
  }
  if ($count <> '1' num) then
    'Error: ' type ': Must be connected to exactly one '
    'Port.' newline
  endif
}

/* Display_Blackboard must be in exactly 1 API call [Message] role */
/* Read_Blackboard must be in exactly 1 API call [Message] role */
/* Read_Sampling_Message must be in exactly 1 API call [Message] role */
/* Receive_Buffer must be in exactly 1 API call [Message] role */
/* Receive_Queuing_Message must be in exactly 1 API call [Message] role */
/* Send_Buffer must be in exactly 1 API call [Message] role */
/* Send_Queuing_Message must be in exactly 1 API call [Message] role */
/* Write_Sampling_Message must be in exactly 1 API call [Message] role */
/* Complement must be in exactly 1 API call [Message] role */
foreach .(Display_Blackboard | Read_Blackboard |
          Read_Sampling_Message | Receive_Buffer |
          Receive_Queuing_Message | Send_Buffer |
          Send_Queuing_Message | Write_Sampling_Message | Complement)
{
  $count = '0'
  do ~API call [Message]
  {
    $count++%null
  }
  if ($count <> '1' num) then
    'Error: ' type ': Must be connected to exactly one '
    'Message.' newline
  endif
```

```
}

/* Conditions on outgoing Control Flow edges of a node must be
   deterministic. */
foreach .(Test Start | Test Fail | Test Pass | Partition | Process |
          Loop | Loop End | Complement | Clear_Blackboard |
          Create_Blackboard | Create_Buffer | Create_Event |
          Create_Process | Create_Queuing_Port |
          Create_Sampling_Port | Create_Semaphore |
          Display_Blackboard | Read_Sampling_Message | Reset_Event |
          Set_Event | Signal_Semaphore | Start | Stop | Stop_Self |
          Write_Sampling_Message | Read_Blackboard | Receive_Buffer |
          Receive_Queuing_Message | Send_Buffer |
          Send_Queuing_Message | Wait_Event | Wait_Semaphore |
          Set_Partition_Mode)
{
  /* more than one unlabeled edge? */
  $count = '0'
  do ~From>Control Flow
  {
    if not :Condition; then
      $count++%null
    endif
  }
  if ($count > '1' num) then
    'Error: ' type ': Must not have more than one unlabeled '
    'outgoing control flow edge.' newline
  endif

  /* return code occurs more than once? */
  $retvals = ' '
  do ~From>Control Flow:Condition
  {
    if $retvals =~ type%wildsp then
      'Error: ' type;1 ': Return code ' type ' must not occur '
      'more than once in outgoing control flow edges.' newline
    else
      variable 'retvals' append type ' ' close
    endif
  }
}


/* Nodes outside a Loop block must not have Control Flow relationships
   to nodes inside a Loop block. */
$loopnodes = ' ';
foreach .Loop
{
  subreport '_findloopnodes' run
}

foreach .(Test Start | Test Fail | Test Pass | Partition | Process |
          Loop End | Complement | Clear_Blackboard |
          Create_Blackboard | Create_Buffer | Create_Event |
          Create_Process | Create_Queuing_Port |
          Create_Sampling_Port | Create_Semaphore |
          Display_Blackboard | Read_Sampling_Message | Reset_Event |
          Set_Event | Signal_Semaphore | Start | Stop | Stop_Self |
          Write_Sampling_Message | Read_Blackboard | Receive_Buffer |
          Receive_Queuing_Message | Send_Buffer |
          Send_Queuing_Message | Wait_Event | Wait_Semaphore |
          Set_Partition_Mode)
{
  if not $loopnodes =~ oid%wildsp then
    /* node not in loop */
    do ~From>Control Flow~To.(*|^Loop|^Loop End)
    {
      if $loopnodes =~ oid%wildsp then
        'Error: Invalid control flow from ' type;1 ' outside of loop '
        'to ' type ' inside loop.' newline
      endif
    }
  endif
}

/* Nodes having a Control Flow relationship to a Loop End block
   must be inside a loop. */
foreach .Loop End
{
  do ~To>Control Flow~From.()
  {
    if not $loopnodes =~ oid%wildsp then
      'Error: ' type ' connected to ' type;1 ' is not inside a loop.'
      newline
    endif
  }
}

endreport


report '_findloopnodes'
  if type <> 'Loop End' then
    if not $loopnodes =~ oid%wildsp then
      variable 'loopnodes' append oid ' ' close
    endif
    do ~From>Control Flow~To.()
    {
      subreport '_findloopnodes' run
    }
  endif
endreport
```

D.3 Test Suite Validator (ITML-B)

```
report '!Validate Test Suite'

/* throw away all output */
to '%null
* $'
endto

/* A Behaviour must be in at least 1 Used by relationship */
foreach .Behaviour
{
  $count = '0'
  do >Used by
  {
    $count++%null
  }
  if ($count < '1' num) then
    'Error: ' type ': Must be connected to at least one '
    'Configuration.' newline
  endif
}

/* A Configuration must be in at least 1 Used by relationship */
foreach .Configuration
{
  $count = '0'
  do >Used by
  {
    $count++%null
  }
  if ($count < '1' num) then
    'Error: ' type ': Must be connected to at least one '
    'Behaviour.' newline
  endif
}

endreport
```

D.4 Test Suite Validator (ITML-C)

```
report '!Validate Test Suite'

/* throw away all output */
to '%null
* $'
endto

/* A Behaviour must be in at least 1 Used by relationship */
foreach .Behaviour
{
  $count = '0'
  do >Used by
  {
    $count++%null
  }
  if ($count < '1' num) then
    'Error: ' type ': Must be connected to at least one '
    'ICD.' newline
  endif
}

/* An ICD must be in at least 1 Used by relationship */
foreach .ICD
{
  $count = '0'
  do >Used by
  {
    $count++%null
  }
  if ($count < '1' num) then
    'Error: ' type ': Must be connected to at least one '
    'Behaviour.' newline
  endif
}

endreport
```

D.5 System Diagram Validator

```
Report '!Validate System Diagram'

/* throw away all output */
to '%null
* $'
endto

/* make into a space-separated wildcard '* xxx *' */
to '%wildsp
/^(.*)$/ $\*\ $1\ \*'
endto

/* SUT must occur exactly 1 time */
$count = '0'
foreach .SUT
{
  $count++%null
}
if ($count <> '1' num) then
  'Error: SUT: Must occur exactly one time.' newline
endif

/* TE must occur exactly 1 time */
$count = '0'
foreach .TE
{
  $count++%null
}
if ($count <> '1' num) then
  'Error: SUT: Must occur exactly one time.' newline
endif

/* The name of each signal and constant definition must be unique */
$names = ' '
foreach .(SUT Input Signal List | SUT Output Signal List | Constant List)
{
  do :List
  {
    $tmp = :Name
    if $names =~ $tmp%wildsp then
      'Error: ' :Name ': Names of signal and constant definitions must '
      'be unique.' newline
    else
      variable 'names' append :Name ' ' close
    endif
  }
}

/* SUT must decompose to either a component diagram or a statechart. */
foreach .SUT
{
  $count = '0'
  do decompositions
  {
    $count++%null
  }
  if ($count <> '1' num) then
    'Error: ' :Name ': SUT must decompose to either a component '
    'diagram or a statechart.' newline
  endif
  do decompositions
  {
    if type = 'Component Diagram' then
      subreport '_validate_decomp' run
    endif
  }
}

endreport


Report '_validate_decomp'

/* A component which is part of the SUT must decompose to either a
   component diagram or a statechart. */
foreach .Component
{
  $count = '0'
  do decompositions
  {
    $count++%null
  }
  if ($count <> '1' num) then
    'Error: ' :Name ': SUT component must decompose to either a '
    'component diagram or a statechart.' newline
  endif
  do decompositions
  {
    if type = 'Component Diagram' then
      subreport '_validate_decomp' run
    endif
  }
}

endreport
```

D.6 Component Diagram Validator

```
Report '!Validate Component Diagram'

/* throw away all output */
to '%null
* $'
endto

/* make into a space-separated wildcard '* xxx *' */
to '%wildsp
/^(.*)$/ $\*\ $1\ \*'
endto

/* Component must occur at least 1 time */
$count = '0'
foreach .Component
{
  $count++%null
}
if ($count < '1' num) then
  'Error: Component: Must occur at least one time.' newline
endif

/* The name of each variable and constant definition must be unique */
$names = ' '
foreach .(Variable List | Constant List)
{
  do :List
  {
    $tmp = :Name
    if $names =~ $tmp%wildsp then
      'Error: ' :Name ': Names of variable and constant definitions must '
      'be unique.' newline
    else
      variable 'names' append :Name ' ' close
    endif
  }
}

endreport
```

D.7 Statechart Validator

```
Report '!Validate Statechart'

/* throw away all output */
to '%null
* $'
endto

/* make into a space-separated wildcard '* xxx *' */
to '%wildsp
/^(.*)$/ $\*\ $1\ \*'
endto

/* Start Location must occur exactly 1 time */
$count = '0'
foreach .Start Location
{
  $count++%null
}
if ($count <> '1' num) then
  'Error: Start Location: Must occur exactly one time.' newline
endif

/* The name of each variable and constant definition must be unique */
$names = ' '
foreach .(Variable List | Constant List)
{
  do :List
  {
    $tmp = :Name
    if $names =~ $tmp%wildsp then
      'Error: ' :Name ': Names of variable and constant definitions must '
      'be unique.' newline
    else
      variable 'names' append :Name ' ' close
    endif
  }
}

endreport
```

APPENDIX E

MERL Intermediate Format Generators

The generators presented in this section are written in MERL (cf. section 3.2.4). They produce the intermediate XML representation (see appendix F) from models built in MetaEdit+.

E.1 Configuration Export Generator

```
report '!Export Configuration'

filename id '.xml' write

/* translate to legal XML text or attribute value */
to '%xml
& $&amp;
< $&lt;
> $&gt;
" $&quot;
        $&#x9;
\
 $&#xD;'
endto

/* convert to lowercase */
to '%lower
A-Z a-z'
endto


'<?xml version="1.0"?>' newline
'<config xmlns="http://www.scarlettproject.eu/tzi/configuration"
    name="' :Name%xml '">' newline
foreach .CPM;
{
'   <cpm name="' :Name%xml '" location="' :Location%xml '">' newline
```

```
  do ~Container>Part of~Containee.Partition;
        orderby :ID; num
  {
'    <partition id="' :ID%xml
        '" period="' :Period%xml '" duration="' :Duration%xml
        '" ramsize="' :RAM Size%xml '">' newline
    do ~Sender>Dataflow~Receiver.();
    {
'      <portgroupref id="' oid%xml '" direction="out"/>' newline
    }
    do ~Receiver>Dataflow~Sender.();
    {
'      <portgroupref id="' oid%xml '" direction="in"/>' newline
    }
'    </partition>' newline
  }

  do ~Container>Part of~Containee.Partition~()>Dataflow~()
        .AFDX Port Group; unique oid
  {
'    <portgroup id="' oid%xml '" type="AFDX"'
        ' characteristic="' :Port Characteristic%lower%xml
        '" size="' :Maximum Message Size%xml
        '" count="' :Count%xml '"/>' newline
  }
  do ~Container>Part of~Containee.Partition~()>Dataflow~()
        .RAM Port Group; unique oid
  {
'    <portgroup id="' oid%xml '" type="RAM"'
        ' characteristic="' :Port Characteristic%lower%xml
        '" size="' :Maximum Message Size%xml
        '" count="' :Count%xml '"/>' newline
  }

'  </cpm>' newline
}
'</config>' newline

close
endreport
```

E.2 Behaviour Export Generator

```
report '!Export Behaviour'

filename id '.xml' write

/* translate to upper case */
to '%upper
a-z A-Z'
endto

/* translate to legal XML text or attribute value */
to '%xml
& $&amp;
< $&lt;
> $&gt;
" $&quot;
         $&#x9;
\
 $&#xD;'
endto

'<?xml version="1.0"?>' newline
'<behaviour xmlns="http://www.scarlettproject.eu/tzi/behaviour"
    name="' :Test Case ID%xml '">' newline

/*** Resources ***/
'  <resources>' newline
foreach .Process Config;
{
'    <process id="' oid %xml '" name="' :Name%xml
        '" stacksize="' :Stack Size%xml '" priority="' :Priority%xml
        '" period="' :Period%xml '" timecapacity="' :Time Capacity%xml
        '" deadline="' :Deadline%xml%upper '"/>' newline
}

foreach .Blackboard
{
'    <blackboard id="' oid%xml '" name="' :Name%xml
        '" maxmsgsize="' :Maximum Message Size%xml '"/>' newline
}

foreach .Buffer
{
'    <buffer id="' oid%xml '" name="' :Name%xml
        '" maxmsgsize="' :Maximum Message Size%xml
        '" maxnbmsg="' :Maximum Number of Messages%xml
        '" queuingdisc="' :Queuing Discipline%xml%upper '"/>' newline
}

foreach .Semaphore
{
'    <semaphore id="' oid%xml '" name="' :Name%xml
        '" currentvalue="' :Current Value%xml
        '" maxvalue="' :Maximum Value%xml
        '" queuingdisc="' :Queuing Discipline%xml%upper '"/>' newline
}

foreach .Event
{
'    <event id="' oid%xml '" name="' :Name%xml '"/>' newline
}

foreach .Port
{
'    <port id="' oid%xml '" type="' :Type%xml '"/>' newline
}

foreach .Message
{
'    <message id="' oid%xml '" pattern="' :Pattern%xml
        '" size="' :Message Size%xml '"/>' newline
}
'  </resources>' newline

/*** Nodes ***/
'  <nodes>' newline
foreach .Test Start;
{
'    <test_start id="' oid%xml '"/>' newline
}

foreach .Test Fail;
{
'    <test_fail id="' oid%xml '"/>' newline
}

foreach .Test Pass;
{
'    <test_pass id="' oid%xml '"/>' newline
}

foreach .Partition
{
'    <partition id="' oid%xml '" name="' :ID%xml '"/>' newline
}

foreach .Process
{
'    <process id="' oid%xml '" name="'
if :Init Process; then
  'INIT'
else
  :Process Config%xml
endif
'"/>' newline
}

foreach .Loop
{
'    <loop id="' oid%xml '"/>' newline
}

foreach .Loop End;
{
'    <loopend id="' oid%xml '"/>' newline
}

foreach .Complement
{
'    <complement id="' oid%xml
        '" operation="' :Operation%xml '"/>' newline
}

foreach .(Clear_Blackboard | Create_Blackboard | Create_Buffer |
          Create_Event | Create_Process | Create_Queuing_Port |
          Create_Sampling_Port | Create_Semaphore |
          Display_Blackboard | Read_Sampling_Message | Reset_Event |
          Set_Event | Signal_Semaphore | Start | Stop | Stop_Self |
          Write_Sampling_Message)
{
'    <' type%xml ' id="' oid%xml '"/>' newline
}
```

```
foreach .(Read_Blackboard | Receive_Buffer | Receive_Queuing_Message |
          Send_Buffer | Send_Queuing_Message | Wait_Event |
          Wait_Semaphore)
{
'    <' type%xml ' id="' oid%xml
        '" timeout="' :Timeout%xml '"/>' newline
}

foreach .Set_Partition_Mode
{
'    <' type%xml ' id="' oid%xml
        '" mode="' :Partition Mode%xml '"/>' newline
}
'  </nodes>' newline

/*** Parameters ***/
'  <parameters>' newline
foreach >Parameter
{
'    <assign op="' do ~(API call [*]).() { oid%xml }
        '" param="' do ~Parameter Object.() { oid%xml } '"/>' newline
}
'  </parameters>' newline

/*** Control Flow ***/
'  <controlflow>' newline
foreach >Control Flow;
{
if :Condition; then
'    <edge from="' do ~From.() { oid%xml }
        '" to="' do ~To.() { oid%xml } '">' newline
do :Condition
{
'      <condition>' type '</condition>' newline
}
'    </edge>' newline
else
'    <edge from="' do ~From.() { oid%xml }
        '" to="' do ~To.() { oid%xml } '"/>' newline
endif
}
'  </controlflow>' newline

'</behaviour>' newline

close
endreport
```

E.3 Test Suite Export Generator (ITML-B)

```
report '!Export Test Suite'

filename id '.xml' write

/* translate to legal XML text or attribute value */
to '%xml
& $&amp;
< $&lt;
> $&gt;
" $&quot;
         $&#x9;
\
 $&#xD;'
endto

'<?xml version="1.0"?>' newline
'<testsuite xmlns="http://www.scarlettproject.eu/tzi/testsuite-b"
    name="' :Name%xml '">' newline

/*** Configurations ***/
'  <configurations>' newline
foreach .Configuration
{
'     <configuration id="' oid%xml '" name="' :Configuration Spec%xml
        '" file="' :Configuration Spec%xml '.xml"/>' newline
}
'  </configurations>' newline

/*** Behaviours ***/
'  <behaviours>' newline
foreach .Behaviour
{
'     <behaviour name="' :Behaviour Spec%xml
        '" file="' :Behaviour Spec%xml '.xml">' newline
  do ~User>Used by~Used.Configuration;
  {
'       <configref id="' oid%xml '"/>' newline
  }
'     </behaviour>' newline
}
'  </behaviours>' newline

'</testsuite>' newline

close
endreport
```

E.4 Test Suite Export Generator (ITML-C)

```
report '!Export Test Suite'

filename id '.xml' write

/* translate to legal XML text or attribute value */
to '%xml
& $&amp;
< $&lt;
> $&gt;
" $&quot;
         $&#x9;
\
 $&#xD;'
```

```
endto

'<?xml version="1.0"?>' newline
'<testsuite xmlns="http://www.scarlettproject.eu/tzi/testsuite-c"
    name="' :Name%xml '">' newline

/*** ICDs ***/
'  <icds>' newline
foreach .ICD
{
'    <icd id="' oid%xml '" path="' :Path%xml '.xml"/>' newline
}
'  </icds>' newline

/*** Behaviours ***/
'  <behaviours>' newline
foreach .Behaviour
{
'    <behaviour name="' :Behaviour Spec%xml
        '" file="' :Behaviour Spec%xml '.xml">' newline
  do ~User>Used by~Used.ICD;
  {
'      <icdref id="' oid%xml '"/>' newline
  }
'    </behaviour>' newline
}
'  </behaviours>' newline

'</testsuite>' newline

close
endreport
```

E.5 System Diagram Export Generator (ITML-A)

```
report '!Export System Diagram'

filename id '.xml' write

/* translate to legal XML text or attribute value */
to '%xml
& $&amp;
< $&lt;
> $&gt;
" $&quot;
         $&#x9;
\
 $&#xD;'
endto

/* indent one level by translating newline to newline + 2 spaces */
to '%indent
\
 $\
\ \ ' endto

/* throw away all output */
to '%null
* $'
endto

'<?xml version="1.0"?>'
newline '<system xmlns="http://www.scarlettproject.eu/tzi/system"
    name="' :Name%xml '">'
```

```
newline '  <sut_inputs>'
foreach .SUT Input Signal List;
{
  do :List
  {
newline '    <signal name="' :Name%xml '" type="' :Variable Type%xml
        '" minimum="' :Minimum%xml '" maximum="' :Maximum%xml
        '" default="' :Default%xml '"/>'
  }
}
newline '  </sut_inputs>'

newline '  <sut_outputs>'
foreach .SUT Output Signal List;
{
  do :List
  {
newline '    <signal name="' :Name%xml '" type="' :Variable Type%xml
        '" minimum="' :Minimum%xml '" maximum="' :Maximum%xml
        '" default="' :Default%xml '"/>'
  }
}
newline '  </sut_outputs>'

newline '  <constants>'
foreach .Constant List;
{
  do :List
  {
newline '    <constant name="' :Name%xml '" type="' :Constant Type%xml
        '" value="' :Value%xml '"/>'
  }
}
newline '  </constants>'

foreach .SUT
{
newline '  <sut name="' :Name%xml '">'
  to '%indent' translate
    do decompositions
    {
      subreport '_sub_' type run
    }
  endto
newline '  </sut>'
}

foreach .TE
{
newline '  <testenv name="' :Name%xml '">'
  $count = '0'
  to '%indent' translate
    do decompositions
    {
      $count++%null
      subreport '_sub_' type run
    }
  endto
  if ($count = '0' num) then
newline '    <variables/>'
newline '    <constants/>'
  endif
newline '  </testenv>'
}

newline '</system>'
newline
```

```
close

endreport


report '_sub_Component Diagram'

newline '  <variables>'
foreach .Variable List;
{
  do :List
  {
newline '    <variable name="' :Name%xml '" type="' :Variable Type%xml
        '" minimum="' :Minimum%xml '" maximum="' :Maximum%xml
        '" default="' :Default%xml '"/>'
  }
}
newline '  </variables>'

newline '  <constants>'
foreach .Constant List;
{
  do :List
  {
newline '    <constant name="' :Name%xml '" type="' :Constant Type%xml
        '" value="' :Value%xml '"/>'
  }
}
newline '  </constants>'

foreach .Component
{
newline '  <component name="' :Name%xml '">'
  to '%indent' translate
    do decompositions
    {
      subreport '_sub_' type run
    }
  endto
newline '  </component>'
}

endreport


report '_sub_Statechart'

/* translate to legal XML text value */
to '%xmltext
& $&amp;
< $&lt;
> $&gt;
" $&quot;'
endto

newline '  <variables>'
foreach .Variable List;
{
  do :List
  {
newline '    <variable name="' :Name%xml '" type="' :Variable Type%xml
        '" minimum="' :Minimum%xml '" maximum="' :Maximum%xml
        '" default="' :Default%xml '"/>'
  }
}
newline '  </variables>'
newline '  <constants>'
foreach .Constant List;
{
  do :List
  {
newline '    <constant name="' :Name%xml '" type="' :Constant Type%xml
        '" value="' :Value%xml '"/>'
  }
}
newline '  </constants>'

newline '  <statechart>'

newline '    <locations>'
foreach .Start Location;
{
newline '      <start_location id="' oid%xml '"/>'
}

foreach .Stop Location;
{
newline '      <stop_location id="' oid%xml '"/>'
}

foreach .Location
{
newline '      <location id="' oid%xml '" name="' :Name%xml '">'
  if :Requirement; then
newline '        <req>' :Requirement%xmltext '</req>'
  endif

  if :Invariant; then
newline '        <invariant>'
    dowhile :Invariant;
    {
      id%xmltext ' '
    }
'</invariant>'
  endif

  if :Entry Action; then
newline '        <entry>'
    do :Entry Action;
    {
      id%xmltext '; '
    }
'</entry>'
  endif

  if :Do Action; then
newline '        <do>'
    do :Do Action;
    {
      id%xmltext '; '
    }
'</do>'
  endif

  if :Exit Action; then
newline '        <exit>'
    do :Exit Action;
    {
      id%xmltext '; '
    }
'</exit>'
  endif
newline '      </location>'
}
newline '    </locations>'

newline '    <transitions>'
foreach >Transition
{
newline '      <transition name="' :Name%xml
        '" from="' do ~Transition_From.() { oid%xml }
        '" to="' do ~Transition_To.() { oid%xml } '">'
  if :Condition; then
newline '        <cond>'
    dowhile :Condition;
    {
      id%xmltext ' '
    }
'</cond>'
  endif

  if :Action; then
newline '        <action>'
    do :Action;
    {
      id%xmltext '; '
    }
'</action>'
  endif
newline '      </transition>'
}
newline '    </transitions>'

newline '  </statechart>'

endreport
```

APPENDIX F

XML Schemata

XML Schema [W3C04] definitions for the intermediate formats produced by the MERL generators (see appendix E) are provided in the following sections.

F.1 Configuration Schema

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
        xmlns="http://www.scarlettproject.eu/tzi/configuration"
        xmlns:c="http://www.scarlettproject.eu/tzi/configuration"
        targetNamespace="http://www.scarlettproject.eu/tzi/configuration"
        elementFormDefault="qualified">
  <xs:element name="config">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="cpm" type="cpmType" minOccurs="0" maxOccurs="unbounded">

              <xs:key name="portgroupKey">
                <xs:selector xpath="c:portgroup"/>
                <xs:field xpath="@id"/>
              </xs:key>

              <xs:keyref name="portgroupKeyRef" refer="portgroupKey">
                <xs:selector xpath="c:partition/c:portgroupref"/>
                <xs:field xpath="@id"/>
              </xs:keyref>

            </xs:element>
      </xs:sequence>
      <xs:attribute name="name" type="xs:string" use="required"/>
    </xs:complexType>
  </xs:element>

  <xs:complexType name="cpmType">
    <xs:choice minOccurs="0" maxOccurs="unbounded">
      <xs:element name="partition" type="partitionType"/>
      <xs:element name="portgroup" type="portgroupType"/>
    </xs:choice>
    <xs:attribute name="name" type="xs:string" use="required"/>
    <xs:attribute name="location" type="xs:string" use="required"/>
  </xs:complexType>

  <xs:complexType name="partitionType">
    <xs:sequence>
      <xs:element name="portgroupref" type="portgrouprefType" minOccurs="0" ↩
           ↪ maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="id" type="xs:string" use="required"/>
    <xs:attribute name="period" type="xs:long" use="required"/>
    <xs:attribute name="duration" type="xs:long" use="required"/>
    <xs:attribute name="ramsize" type="xs:unsignedInt" use="required"/>
  </xs:complexType>

  <xs:complexType name="portgroupType">
    <xs:attribute name="id" type="xs:string" use="required"/> <!-- key -->
    <xs:attribute name="type" type="porttypeType" use="required"/>
    <xs:attribute name="characteristic" type="characteristicType" use="required"/>
    <xs:attribute name="size" type="xs:unsignedInt" use="required"/>
    <xs:attribute name="count" type="xs:unsignedInt" use="required"/>
  </xs:complexType>

  <xs:complexType name="portgrouprefType">
    <xs:attribute name="id" type="xs:string" use="required"/> <!-- ref -->
    <xs:attribute name="direction" type="directionType" use="required"/>
  </xs:complexType>

  <xs:simpleType name="porttypeType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="AFDX"/>
      <xs:enumeration value="RAM"/>
    </xs:restriction>
  </xs:simpleType>

  <xs:simpleType name="characteristicType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="sampling"/>
      <xs:enumeration value="queuing"/>
    </xs:restriction>
  </xs:simpleType>

  <xs:simpleType name="directionType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="in"/>
      <xs:enumeration value="out"/>
    </xs:restriction>
  </xs:simpleType>
</xs:schema>
```

F.2 Behaviour Schema

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
        xmlns="http://www.scarlettproject.eu/tzi/behaviour"
        xmlns:b="http://www.scarlettproject.eu/tzi/behaviour"
        targetNamespace="http://www.scarlettproject.eu/tzi/behaviour"
        elementFormDefault="qualified">
  <xs:element name="behaviour">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="resources" type="resourcesType"/>
        <xs:element name="nodes" type="nodesType"/>
        <xs:element name="parameters" type="parametersType"/>
      <xs:element name=”controlflow” type=”controlflowType”/>
    </xs:sequence>
    <xs:attribute name=”name” type=”xs:string” use=”required”/>
  </xs:complexType>

  <xs:key name=”resourcesKey”>
    <xs:selector xpath=”b:resources/b:*”/>
    <xs:field xpath=”@id”/>
  </xs:key>
  <xs:key name=”nodesKey”>
    <xs:selector xpath=”b:nodes/b:*”/>
    <xs:field xpath=”@id”/>
  </xs:key>

  <xs:keyref name=”resourcesKeyRef” refer=”resourcesKey”>
    <xs:selector xpath=”b:parameters/b:assign”/>
    <xs:field xpath=”@param”/>
  </xs:keyref>
  <xs:keyref name=”operationsKeyRef” refer=”nodesKey”>
    <xs:selector xpath=”b:parameters/b:assign”/>
    <xs:field xpath=”@op”/>
  </xs:keyref>
  <xs:keyref name=”nodesFromKeyRef” refer=”nodesKey”>
    <xs:selector xpath=”b:controlflow/b:edge”/>
    <xs:field xpath=”@from”/>
  </xs:keyref>
  <xs:keyref name=”nodesToKeyRef” refer=”nodesKey”>
    <xs:selector xpath=”b:controlflow/b:edge”/>
    <xs:field xpath=”@to”/>
  </xs:keyref>
</xs:element>

<!-- Resources -->
<xs:complexType name=”resourcesType”>
  <xs:choice minOccurs=”0” maxOccurs=”unbounded”>
    <xs:element name=”process” type=”processConfigType”/>
    <xs:element name=”blackboard” type=”blackboardType”/>
    <xs:element name=”buffer” type=”bufferType”/>
    <xs:element name=”port” type=”portType”/>
    <xs:element name=”semaphore” type=”semaphoreType”/>
    <xs:element name=”event” type=”namedResourceType”/>
    <xs:element name=”message” type=”messageType”/>
  </xs:choice>
</xs:complexType>

<xs:complexType name=”resourceType”>
  <xs:attribute name=”id” type=”xs:string” use=”required”/> <!-- key -->
</xs:complexType>

<xs:complexType name=”namedResourceType”>
  <xs:complexContent>
    <xs:extension base=”resourceType”>
      <xs:attribute name=”name” type=”xs:string” use=”required”/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name=”processConfigType”>
  <xs:complexContent>
    <xs:extension base=”namedResourceType”>
      <xs:attribute name=”stacksize” type=”xs:unsignedInt” use=”required”/>
      <xs:attribute name=”priority” type=”xs:int” use=”required”/>
      <xs:attribute name=”period” type=”xs:long” use=”required”/>
      <xs:attribute name=”timecapacity” type=”xs:long” use=”required”/>
      <xs:attribute name=”deadline” type=”deadlineType” use=”required”/>
    </xs:extension>
  </xs:complexContent>
```

```
</xs:complexType>

<xs:complexType name="blackboardType">
  <xs:complexContent>
    <xs:extension base="namedResourceType">
      <xs:attribute name="maxmsgsize" type="xs:unsignedInt" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="bufferType">
  <xs:complexContent>
    <xs:extension base="namedResourceType">
      <xs:attribute name="maxmsgsize" type="xs:unsignedInt" use="required"/>
      <xs:attribute name="maxnbmsg" type="xs:unsignedInt" use="required"/>
      <xs:attribute name="queuingdisc" type="queuingDisciplineType" ↩
           ↪ use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="portType">
  <xs:complexContent>
    <xs:extension base="resourceType">
      <xs:attribute name="type" type="portItemType" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="semaphoreType">
  <xs:complexContent>
    <xs:extension base="namedResourceType">
      <xs:attribute name="currentvalue" type="xs:unsignedInt" use="required"/>
      <xs:attribute name="maxvalue" type="xs:unsignedInt" use="required"/>
      <xs:attribute name="queuingdisc" type="queuingDisciplineType" ↩
           ↪ use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="messageType">
  <xs:complexContent>
    <xs:extension base="resourceType">
      <xs:attribute name="pattern" type="xs:string" use="required"/>
      <xs:attribute name="size" type="xs:unsignedInt" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<!-- Nodes -->
<xs:complexType name="nodesType">
  <xs:choice minOccurs="0" maxOccurs="unbounded">
    <xs:element name="test_start" type="nodeType"/>
    <xs:element name="test_fail" type="nodeType"/>
    <xs:element name="test_pass" type="nodeType"/>
    <xs:element name="partition" type="namedNodeType"/>
    <xs:element name="process" type="namedNodeType"/>
    <xs:element name="loop" type="nodeType"/>
    <xs:element name="loopend" type="nodeType"/>
    <xs:element name="complement" type="complementNodeType"/>
    <xs:element name="Clear_Blackboard" type="nodeType"/>
    <xs:element name="Create_Blackboard" type="nodeType"/>
    <xs:element name="Create_Buffer" type="nodeType"/>
    <xs:element name="Create_Event" type="nodeType"/>
    <xs:element name="Create_Process" type="nodeType"/>
    <xs:element name="Create_Queuing_Port" type="nodeType"/>
    <xs:element name="Create_Sampling_Port" type="nodeType"/>
    <xs:element name="Create_Semaphore" type="nodeType"/>
    <xs:element name="Display_Blackboard" type="nodeType"/>
    <xs:element name="Read_Sampling_Message" type="nodeType"/>
    <xs:element name="Reset_Event" type="nodeType"/>
    <xs:element name="Set_Event" type="nodeType"/>
    <xs:element name="Signal_Semaphore" type="nodeType"/>
    <xs:element name="Start" type="nodeType"/>
    <xs:element name="Stop" type="nodeType"/>
    <xs:element name="Stop_Self" type="nodeType"/>
    <xs:element name="Write_Sampling_Message" type="nodeType"/>
    <xs:element name="Read_Blackboard" type="timeoutNodeType"/>
    <xs:element name="Receive_Buffer" type="timeoutNodeType"/>
    <xs:element name="Receive_Queuing_Message" type="timeoutNodeType"/>
    <xs:element name="Send_Buffer" type="timeoutNodeType"/>
    <xs:element name="Send_Queuing_Message" type="timeoutNodeType"/>
    <xs:element name="Wait_Event" type="timeoutNodeType"/>
    <xs:element name="Wait_Semaphore" type="timeoutNodeType"/>
    <xs:element name="Set_Partition_Mode" type="setPartitionModeNodeType"/>
  </xs:choice>
</xs:complexType>

<xs:complexType name="nodeType">
  <xs:attribute name="id" type="xs:string" use="required"/> <!-- key -->
</xs:complexType>

<xs:complexType name="namedNodeType">
  <xs:complexContent>
    <xs:extension base="nodeType">
      <xs:attribute name="name" type="xs:string" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="timeoutNodeType">
  <xs:complexContent>
    <xs:extension base="nodeType">
      <xs:attribute name="timeout" type="xs:long" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="complementNodeType">
  <xs:complexContent>
    <xs:extension base="nodeType">
      <xs:attribute name="operation" type="complementOperationType" ↩
          ↪ use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="setPartitionModeNodeType">
  <xs:complexContent>
    <xs:extension base="nodeType">
      <xs:attribute name="mode" type="partitionModeType" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<!-- Parameters -->
<xs:complexType name="parametersType">
  <xs:sequence>
    <xs:element name="assign" type="assignType" minOccurs="0" ↩
        ↪ maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="assignType">
```

```
    <xs:attribute name="op" type="xs:string" use="required"/> <!-- reference -->
    <xs:attribute name="param" type="xs:string" use="required"/> <!-- reference -->
  </xs:complexType>

  <!-- Control Flow -->
  <xs:complexType name="controlflowType">
    <xs:sequence>
      <xs:element name="edge" type="edgeType" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="edgeType">
    <xs:sequence>
      <xs:element name="condition" type="xs:string" minOccurs="0" ↩
           ↪ maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="from" type="xs:string" use="required"/> <!-- reference -->
    <xs:attribute name="to" type="xs:string" use="required"/> <!-- reference -->
  </xs:complexType>

  <!-- enumeration types -->
  <xs:simpleType name="deadlineType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="SOFT"/>
      <xs:enumeration value="HARD"/>
    </xs:restriction>
  </xs:simpleType>

  <xs:simpleType name="queuingDisciplineType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="FIFO"/>
      <xs:enumeration value="PRIORITY"/>
    </xs:restriction>
  </xs:simpleType>

  <xs:simpleType name="portItemType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="AFDX Queuing Output Port"/>
      <xs:enumeration value="AFDX Queuing Input Port"/>
      <xs:enumeration value="AFDX Sampling Output Port"/>
      <xs:enumeration value="AFDX Sampling Input Port"/>
      <xs:enumeration value="RAM Queuing Output Port"/>
      <xs:enumeration value="RAM Queuing Input Port"/>
      <xs:enumeration value="RAM Sampling Output Port"/>
      <xs:enumeration value="RAM Sampling Input Port"/>
    </xs:restriction>
  </xs:simpleType>

  <xs:simpleType name="complementOperationType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="Read"/>
      <xs:enumeration value="Write"/>
    </xs:restriction>
  </xs:simpleType>

  <xs:simpleType name="partitionModeType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="IDLE"/>
      <xs:enumeration value="COLD_START"/>
      <xs:enumeration value="WARM_START"/>
      <xs:enumeration value="NORMAL"/>
    </xs:restriction>
  </xs:simpleType>

</xs:schema>
```

F.3 Test Suite Schema (ITML-B)

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
        xmlns="http://www.scarlettproject.eu/tzi/testsuite-b"
        xmlns:t="http://www.scarlettproject.eu/tzi/testsuite-b"
        targetNamespace="http://www.scarlettproject.eu/tzi/testsuite-b"
        elementFormDefault="qualified">
  <xs:element name="testsuite">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="configurations" minOccurs="0" maxOccurs="1">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="configuration" type="configurationType" ↩
                   ↪ minOccurs="0" maxOccurs="unbounded"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <xs:element name="behaviours">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="behaviour" type="behaviourType" minOccurs="0" ↩
                   ↪ maxOccurs="unbounded"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
      <xs:attribute name="name" type="xs:string" use="required"/>
    </xs:complexType>

    <xs:key name="configurationKey">
      <xs:selector xpath="t:configurations/t:configuration"/>
      <xs:field xpath="@id"/>
    </xs:key>

    <xs:keyref name="configurationKeyRef" refer="configurationKey">
      <xs:selector xpath="t:behaviours/t:behaviour/t:configref"/>
      <xs:field xpath="@id"/>
    </xs:keyref>
  </xs:element>

  <xs:complexType name="configurationType">
    <xs:attribute name="id" type="xs:string" use="required"/> <!-- key -->
    <xs:attribute name="name" type="xs:string" use="required"/>
    <xs:attribute name="file" type="xs:string" use="required"/>
  </xs:complexType>

  <xs:complexType name="behaviourType">
    <xs:sequence>
      <xs:element name="configref" type="configrefType" minOccurs="0" ↩
           ↪ maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="name" type="xs:string" use="required"/>
    <xs:attribute name="file" type="xs:string" use="required"/>
  </xs:complexType>

  <xs:complexType name="configrefType">
    <xs:attribute name="id" type="xs:string" use="required"/> <!-- ref -->
  </xs:complexType>
</xs:schema>
```

F.4 Test Suite Schema (ITML-C)

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
        xmlns="http://www.scarlettproject.eu/tzi/testsuite-c"
        xmlns:t="http://www.scarlettproject.eu/tzi/testsuite-c"
        targetNamespace="http://www.scarlettproject.eu/tzi/testsuite-c"
        elementFormDefault="qualified">
  <xs:element name="testsuite">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="icds" minOccurs="0" maxOccurs="1">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="icd" type="icdType" minOccurs="0" ↩
                   ↪ maxOccurs="unbounded"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <xs:element name="behaviours">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="behaviour" type="behaviourType" minOccurs="0" ↩
                   ↪ maxOccurs="unbounded"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
      <xs:attribute name="name" type="xs:string" use="required"/>
    </xs:complexType>

    <xs:key name="icdKey">
      <xs:selector xpath="t:icds/t:icd"/>
      <xs:field xpath="@id"/>
    </xs:key>

    <xs:keyref name="icdKeyRef" refer="icdKey">
      <xs:selector xpath="t:behaviours/t:behaviour/t:icdref"/>
      <xs:field xpath="@id"/>
    </xs:keyref>
  </xs:element>

  <xs:complexType name="icdType">
    <xs:attribute name="id" type="xs:string" use="required"/> <!-- key -->
    <xs:attribute name="path" type="xs:string" use="required"/>
  </xs:complexType>

  <xs:complexType name="behaviourType">
    <xs:sequence>
      <xs:element name="icdref" type="icdrefType" minOccurs="0" ↩
           ↪ maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="name" type="xs:string" use="required"/>
    <xs:attribute name="file" type="xs:string" use="required"/>
  </xs:complexType>

  <xs:complexType name="icdrefType">
    <xs:attribute name="id" type="xs:string" use="required"/> <!-- ref -->
  </xs:complexType>
</xs:schema>
```

F.5 System Diagram Schema (ITML-A)

```
<?xml version=”1.0”?>
<xs:schema xmlns:xs=”http://www.w3.org/2001/XMLSchema”
    xmlns=”http://www.scarlettproject.eu/tzi/system”
    xmlns:s=”http://www.scarlettproject.eu/tzi/system”
    targetNamespace=”http://www.scarlettproject.eu/tzi/system”
    elementFormDefault=”qualified”>
  <xs:element name=”system”>
    <xs:complexType>
      <xs:sequence>
        <xs:element name=”sut_inputs” type=”signalsListType”/>
        <xs:element name=”sut_outputs” type=”signalsListType”/>
        <xs:element name=”constants” type=”constantsListType”/>
        <xs:element name=”sut” type=”componentType”/>
        <xs:element name=”testenv” type=”componentType”/>
      </xs:sequence>
      <xs:attribute name=”name” type=”xs:string” use=”required”/>
    </xs:complexType>
  </xs:element>

  <xs:complexType name=”signalsListType”>
    <xs:sequence>
      <xs:element name=”signal” type=”variableType” minOccurs=”0” ↩
          ↪ maxOccurs=”unbounded”/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name=”constantsListType”>
    <xs:sequence>
      <xs:element name=”constant” type=”constantType” minOccurs=”0” ↩
          ↪ maxOccurs=”unbounded”/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name=”variablesListType”>
    <xs:sequence>
      <xs:element name=”variable” type=”variableType” minOccurs=”0” ↩
          ↪ maxOccurs=”unbounded”/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name=”variableType”>
    <xs:attribute name=”name” type=”xs:string” use=”required”/>
    <xs:attribute name=”type” type=”datatypeType” use=”required”/>
    <xs:attribute name=”minimum” type=”valueType” use=”required”/>
    <xs:attribute name=”maximum” type=”valueType” use=”required”/>
    <xs:attribute name=”default” type=”valueType” use=”required”/>
  </xs:complexType>

  <xs:complexType name=”constantType”>
    <xs:attribute name=”name” type=”xs:string” use=”required”/>
    <xs:attribute name=”type” type=”datatypeType” use=”required”/>
    <xs:attribute name=”value” type=”valueType” use=”required”/>
  </xs:complexType>

  <xs:simpleType name=”datatypeType”>
    <xs:restriction base=”xs:string”>
      <xs:enumeration value=”bool”/>
      <xs:enumeration value=”char”/>
      <xs:enumeration value=”int”/>
      <xs:enumeration value=”float”/>
      <xs:enumeration value=”timer”/>
    </xs:restriction>
  </xs:simpleType>
```

```
<xs:simpleType name="valueType">
  <xs:union memberTypes="xs:decimal boolType"/>
</xs:simpleType>

<xs:simpleType name="boolType">
  <xs:restriction base="xs:string">
    <xs:enumeration value="False"/>
    <xs:enumeration value="True"/>
  </xs:restriction>
</xs:simpleType>


<xs:complexType name="componentType">
  <xs:sequence>
    <xs:element name="variables" type="variablesListType"/>
    <xs:element name="constants" type="constantsListType"/>
    <xs:choice minOccurs="0">
      <xs:element name="component" type="componentType" maxOccurs="unbounded"/>
      <xs:element name="statechart" type="statechartType">

        <xs:key name="locationsKey">
          <xs:selector xpath="s:locations/s:*"/>
          <xs:field xpath="@id"/>
        </xs:key>

        <xs:keyref name="locationsFromKeyRef" refer="locationsKey">
          <xs:selector xpath="s:transitions/s:transition"/>
          <xs:field xpath="@from"/>
        </xs:keyref>
        <xs:keyref name="locationsToKeyRef" refer="locationsKey">
          <xs:selector xpath="s:transitions/s:transition"/>
          <xs:field xpath="@to"/>
        </xs:keyref>

      </xs:element>
    </xs:choice>
  </xs:sequence>
  <xs:attribute name="name" type="xs:string" use="required"/>
</xs:complexType>

<xs:complexType name="statechartType">
  <xs:sequence>
    <xs:element name="locations" type="locationsListType"/>
    <xs:element name="transitions" type="transitionsListType"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="locationsListType">
  <xs:sequence>
    <xs:element name="start_location" type="unnamedLocationType" minOccurs="1" ↩
         ↪ maxOccurs="1"/>
    <xs:element name="stop_location" type="unnamedLocationType" minOccurs="0" ↩
         ↪ maxOccurs="unbounded"/>
    <xs:element name="location" type="namedLocationType" minOccurs="0" ↩
         ↪ maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="transitionsListType">
  <xs:sequence>
    <xs:element name="transition" type="transitionType" minOccurs="0" ↩
         ↪ maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="unnamedLocationType">
  <xs:attribute name="id" type="xs:string" use="required"/> <!-- key -->
  </xs:complexType>

  <xs:complexType name="namedLocationType">
    <xs:complexContent>
      <xs:extension base="unnamedLocationType">
        <xs:sequence>
          <xs:element name="req" type="xs:string" minOccurs="0"/>
          <xs:element name="entry" type="xs:string" minOccurs="0"/>
          <xs:element name="do" type="xs:string" minOccurs="0"/>
          <xs:element name="exit" type="xs:string" minOccurs="0"/>
        </xs:sequence>
        <xs:attribute name="name" type="xs:string" use="required"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>

  <xs:complexType name="transitionType">
    <xs:sequence>
      <xs:element name="req" type="xs:string" minOccurs="0"/>
      <xs:element name="cond" type="xs:string" minOccurs="0"/>
      <xs:element name="action" type="xs:string" minOccurs="0"/>
    </xs:sequence>
    <xs:attribute name="id" type="xs:string" use="required"/>
    <xs:attribute name="name" type="xs:string" use="required"/>
    <xs:attribute name="from" type="xs:string" use="required"/> <!-- reference -->
    <xs:attribute name="to" type="xs:string" use="required"/>  <!-- reference -->
  </xs:complexType>

</xs:schema>
```

APPENDIX G

List of Acronyms

AFDX	Avionics Full Duplex Switched Ethernet
API	Application Programming Interface
ARINC	Aeronautical Radio, Inc.
AST	Abstract Syntax Tree
BAG	Bandwidth Allocation Gap
CAN	Controller Area Network
CPM	Core Processing Module
CSV	Comma-Separated Values
DME	Distributed Modular Electronics
DS	Data Set
DSL	Domain-specific Language
DSM	Domain-specific Modelling
DSML	Domain-specific Modelling Language
ETSI	European Telecommunications Standards Institute
FDS	Functional Data Set
FS	Functional Status
FSS	Functional Status Set
FSD	Fire and Smoke Detection
ICD	Interface Control Document
IFG	Intermediate Format Generator
IMA	Integrated Modular Avionics
IMA2G	Second-generation IMA
IMR	Intermediate Model Representation

ITML	IMA Test Modelling Language
MBT	Model-based testing
MERL	MetaEdit+ Reporting Language
MTC	Main Test Component
OS	Operating System
PA	Platform Adapter
PDU	Protocol Data Unit
RDC	Remote Data Concentrator
REU	Remote Electronics Unit
RPC	Remote Power Controller
SA	SUT Adapter
SD	Smoke Detector
SSM	Sign/Status Matrix
SUT	System Under Test
TA	Test Agent
TACP	Test Agent Control Protocol
TCI	TTCN-3 Control Interface
TCP	Transmission Control Protocol
TE	Test Executable
TRI	TTCN-3 Runtime Interface
TSI	Test System Interface
TTCN-3	Testing and Test Control Notation Version 3
UDP	User Datagram Protocol
UML	Unified Modeling Language
VL	Virtual Link
XDR	External Data Representation
XML	Extensible Markup Language

Bibliography

[Aer04] Aeronautical Radio, Inc. *ARINC Specification 429P1-17 Mark 33: Digital Information Transfer System (DITS), Part 1, Functional Description, Electrical Interface, Label Assignments and Word Formats*, May 2004.

[Aer05] Aeronautical Radio, Inc. *ARINC Specification 653P1-2: Avionics Application Software Standard Interface, Part 1 – Required Services*, December 2005.

[Aer06] Aeronautical Radio, Inc. *ARINC Specification 653P3: Avionics Application Software Standard Interface, Part 3 – Conformity Test Specification*, October 2006.

[Aer07a] Aeronautical Radio, Inc. *ARINC Report 615A-3: Software Data Loader Using Ethernet Interface*, June 2007.

[Aer07b] Aeronautical Radio, Inc. *ARINC Specification 653P2: Avionics Application Software Standard Interface, Part 2 – Extended Services*, January 2007.

[Aer09] Aeronautical Radio, Inc. *ARINC Specification 664P7-1: Aircraft Data Network, Part 7, Avionics Full-Duplex Switched Ethernet*, September 2009.

[AUT] AUTOSAR. Partnership website [online]. URL: `http://www.autosar.org/`.

[BDG+08] Paul Baker, Zhen Ru Dai, Jens Grabowski, Øystein Haugen, Ina Schieferdecker, and Clay Williams. *Model-Driven Testing Using the UML Testing Profile*. Springer, Berlin, Heidelberg, 2008.

[Bin12] Robert Binder. *2011 Model-based Testing User Survey: Results and Analysis*. System Verification Associates, LLC, February 2012. URL: `http://robertvbinder.com/wp-content/uploads/rvb-pdf/arts/MBT-User-Survey.pdf`.

[CST] Code Synthesis Tools CC. XSD: XML Data Binding for C++. Product website [online]. URL: http://www.codesynthesis.com/products/xsd/.

[EH08] Christof Efkemann and Tobias Hartmann. Specification of conditions for error diagnostics. *Electronic Notes in Theoretical Computer Science*, 217:97 – 112, 2008. Proceedings of the 3rd International Workshop on Systems Software Verification (SSV 2008). URL: http://dx.doi.org/10.1016/j.entcs.2008.06.044.

[EP11] Christof Efkemann and Jan Peleska. Model-based testing for the second generation of integrated modular avionics. *IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pages 55–62, 2011. URL: http://doi.ieeecomputersociety.org/10.1109/ICSTW.2011.72.

[ETS] ETSI. TTCN-3 website [online]. URL: http://www.ttcn-3.org/.

[ETS07] ETSI. *Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 3: TTCN-3 Graphical presentation Format (GFT)*, February 2007. ETSI ES 201 873-3.

[ETS09a] ETSI. *Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language*, June 2009. ETSI ES 201 873-1.

[ETS09b] ETSI. *Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 4: Operational Semantics*, June 2009. ETSI ES 201 873-4.

[ETS09c] ETSI. *Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 5: TTCN-3 Runtime Interface (TRI)*, June 2009. ETSI ES 201 873-5.

[ETS09d] ETSI. *Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 6: TTCN-3 Control Interface (TCI)*, June 2009. ETSI ES 201 873-6.

[Feu13] Johannes Feuser. *Open Source Software for Train Control Applications and its Architectural Implications*. PhD thesis, University of Bremen, 2013.

[Fil03] Bill Filmer. Open systems avionics architectures considerations. *Aerospace and Electronic Systems Magazine, IEEE*, 18(9):3–10, September 2003. doi:10.1109/MAES.2003.1232153.

[FP10] Martin Fowler and Rebecca Parsons. *Domain-Specific Languages*. Addison-Wesley Longman, Amsterdam, September 2010.

[FP7] Publications Office of the European Union. Website of the Seventh Framework Programme [online]. URL: `http://cordis.europa.eu/fp7/`.

[GP96] Thomas R. G. Green and Marian Petre. Usability analysis of visual programming environments: A 'cognitive dimensions' framework. *Journal of Visual Languages & Computing*, 7(2):131–174, 1996. URL: `http://dx.doi.org/10.1006/jvlc.1996.0009`.

[Har87] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987. doi:10.1016/0167-6423(87)90035-9.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, August 1978. URL: `http://doi.acm.org/10.1145/359576.359585`.

[Hui11] Antti Huima. Behavioral system models versus models of testing strategies in functional test generation. In *Model-Based Testing for Embedded Systems*. CRC Press, Boca Raton, September 2011.

[Kel95] Steven Kelly. What's in a relationship? On distinguishing property holding and object binding. In Eckhard D. Falkenberg and Wolfgang Hesse, editors, *Proceedings of the IFIP international working conference on Information system concepts: Towards a consolidation of views*, pages 144–159, London, UK, 1995. Chapman & Hall, Ltd.

[Kel97] Steven Kelly. *Towards a comprehensive MetaCASE and CAME environment*. PhD thesis, University of Jyväskylä, 1997.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall Professional Technical Reference, second edition, 1988.

[KT08] Steven Kelly and Juha-Pekka Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation*. John Wiley & Sons, Inc., Hoboken, NJ, April 2008.

[LP10] Helge Löding and Jan Peleska. Timed moore automata: Test data generation and model checking. In *Third International Conference on Software Testing, Verification, and*

Validation, pages 449–458, 2010. URL: http://doi.ieeecomputersociety.org/10.1109/ICST.2010.60.

[MC] MetaCase. MetaEdit+ product website [online]. URL: http://www.metacase.com/products.html.

[Mew10] Kirsten Mewes. *Domain-specific Modelling of Railway Control Systems with Integrated Verification and Validation*. PhD thesis, University of Bremen, March 2010.

[MWUG] MetaCase. MetaEdit+ Version 4.5 Workbench User's Guide [online]. URL: http://www.metacase.com/support/45/manuals/mwb/Mw.html.

[OH05] Aliki Ott and Tobias Hartmann. Domain specific V&V strategies for aircraft applications. In *6th ICSTEST International Conference on Software Testing*, Düsseldorf, April 2005.

[OMG11] Object Management Group. *Unified Modeling Language, Superstructure*, August 2011. URL: http://www.omg.org/spec/UML/2.4.1/Superstructure.

[OMG13] Object Management Group. *UML Testing Profile (UTP)*, April 2013. URL: http://www.omg.org/spec/UTP/1.2/.

[ORC] Oracle Corporation. Java language keywords [online]. URL: http://docs.oracle.com/javase/tutorial/java/nutsandbolts/_keywords.html.

[Ott07] Aliki Ott. *System Testing in the Avionics Domain*. PhD thesis, University of Bremen, December 2007.

[Pel02] Jan Peleska. Formal methods for test automation – hard real-time testing of controllers for the Airbus aircraft family. In *Proc. of the Sixth Biennial World Conference on Integrated Design & Process Technology (IDPT2002), Pasadena, California, June 23–28, 2002*. Society for Design and Process Science, June 2002. ISSN 1090-9389.

[Pel13] Jan Peleska. Industrial-strength model-based testing – state of the art and current challenges. In *Proceedings of the Eighth Workshop on Model-Based Testing*, pages 3–28, 2013. URL: http://dx.doi.org/10.4204/EPTCS.111.1.

[PHL$^+$11] Jan Peleska, Artur Honisch, Florian Lapschies, Helge Löding, Hermann Schmid, Peer Smuda, Elena Vorobev, and Cornelia Zahlten. A real-world benchmark model for testing concurrent real-time systems in the automotive domain. In *Proceedings of the 23rd IFIP WG 6.1 international conference on Testing* *software and systems*, ICTSS'11, pages 146–161, Berlin, Heidelberg, 2011. Springer-Verlag. URL: http://dl.acm.org/citation.cfm?id=2075545.2075556.

[Plo81] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.

[PPW+05] A. Pretschner, W. Prenninger, S. Wagner, C. Kühnel, M. Baumgartner, B. Sostawa, R. Zölch, and T. Stauner. One evaluation of model-based testing and its automation. In *Proceedings of the 27th international conference on Software engineering*, ICSE '05, pages 392–401, New York, NY, USA, 2005. ACM. URL: http://doi.acm.org/10.1145/1062455.1062529.

[PVL11] Jan Peleska, Elena Vorobev, and Florian Lapschies. Automated test case generation with SMT-solving and abstract interpretation. In *NASA Formal Methods*, volume 6617 of *Lecture Notes in Computer Science*, pages 298–312. Springer, 2011. URL: http://dx.doi.org/10.1007/978-3-642-20398-5_22.

[RT06] S. Ranise and C. Tinelli. Satisfiability modulo theories. *Trends and Controversies–IEEE Intelligent Systems Magazine*, 21(6):71–81, 2006.

[RTC92] RTCA, Inc. *DO-178B: Software Considerations in Airborne Systems and Equipment Certification*, December 1992.

[RTC01] RTCA, Inc. *DO-248B: Final Annual Report For Clarification Of DO-178B "Software Considerations in Airborne Systems and Equipment Certification"*, October 2001.

[RTC05] RTCA, Inc. *DO-297: Integrated Modular Avionics (IMA) Development Guidance and Certification Considerations*, November 2005.

[SCA] SCARLETT Consortium. Project website [online]. URL: http://www.scarlettproject.eu/.

[Sch00] S. Schneider. *Concurrent and Real-time Systems – The CSP Approach*. John Wiley & Sons, Ltd, Chichester, 2000.

[SM87] Sun Microsystems, Inc. *XDR: External Data Representation Standard*. Internet Engineering Task Force, June 1987. RFC 1014. URL: http://www.ietf.org/rfc/rfc1014.txt.

[SMD+08] Diana Alina Serbanescu, Victoria Molovata, George Din, Ina Schieferdecker, and Ilja Radusch. Real-time testing with

TTCN-3. In Kenji Suzuki, Teruo Higashino, Andreas Ulrich, and Toru Hasegawa, editors, *Testing of Software and Communicating Systems*, volume 5047 of *Lecture Notes in Computer Science*, pages 283–301. Springer, Berlin, Heidelberg, 2008. URL: http://dx.doi.org/10.1007/978-3-540-68524-1_20.

[SP13] Bernard Stepien and Liam Peyton. Challenges of testing periodic messages in avionics systems using TTCN-3. In Hüsnü Yenigün, Cemal Yilmaz, and Andreas Ulrich, editors, *Testing Software and Systems*, volume 8254 of *Lecture Notes in Computer Science*, pages 207–222. Springer, Berlin, Heidelberg, 2013. URL: http://dx.doi.org/10.1007/978-3-642-41707-8_14.

[TTW] Testing Technologies IST GmbH. TTworkbench product website [online]. URL: http://www.testingtech.com/products/ttworkbench.php.

[UL07] Mark Utting and Bruno Legeard. *Practical Model-Based Testing. A Tools Approach*. Morgan Kaufman, San Francisco, March 2007.

[W3C04] World Wide Web Consortium. *XML Schema Part 0: Primer Second Edition*, October 2004. URL: http://www.w3.org/TR/xmlschema-0/.

[WDT⁺05] Colin Wilcock, Thomas Deiß, Stephan Tobies, Stefan Keil, Federico Engler, and Stephan Schulz. *An Introduction to TTCN-3*, chapter 12. John Wiley & Sons, Ltd, Chichester, April 2005.

[Wei10] Stephan Weißleder. *Test Models and Coverage Criteria for Automatic Model-Based Test Generation with UML State Machines*. PhD thesis, Humboldt-Universität zu Berlin, 2010.

[Wil04] David Wile. Lessons learned from real DSL experiments. *Science of Computer Programming*, 51(3):265–290, June 2004. URL: http://dx.doi.org/10.1016/j.scico.2003.12.006.

[ZSM11a] Justyna Zander, Ina Schieferdecker, and Pieter J. Mosterman, editors. *Model-Based Testing for Embedded Systems*. CRC Press, Boca Raton, September 2011.

[ZSM11b] Justyna Zander, Ina Schieferdecker, and Pieter J. Mosterman. A taxonomy of model-based testing for embedded systems from multiple industry domains. In *Model-Based Testing for Embedded Systems*. CRC Press, Boca Raton, September 2011.

Note: Hyperlinks in the bibliography were correct as of 10th June 2014.