Project no.: **IST-FP6-STREP-26979**

Project full title: **Highly dependable ip-based networks and services**

Project Acronym: **HIDENETS**

Deliverable no.: D5.3

Title of the deliverable: Refined design and testing framework, methodology and application results

Contractual Date of Delivery to the CEC:	**31st Dec. 2008**
Actual Date of Delivery to the CEC:	**xx xxx xx**
Organisation name of lead contractor for this deliverable:	**P02 BME**
Author(s): Gábor Huszerl[P02] and Hélène Waeselynck [P06] (ed.), Zoltán Égel [P02], András Kövi [P02], Zoltán Micskei [P02], Minh Duc N'Guyen [P06], Gergely Pintér [P02] and Nicolas Rivière [P06]	
Participants(s): P02 BME, P05 FSC, P06 LAAS-CNRS	
Work package contributing to the deliverable:	**WP5**
Nature:	**R/P**
Version:	**1.0**
Total number of pages:	**118**
Start date of project:	1st Jan. 2006 **Duration:** 36 month

Project co-funded by the European Commission within the Sixth Framework Programme (2002-2006)		
Dissemination Level		
PU	Public	**X**
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

Abstract:

This Deliverable summarises the results of the HIDENETS project related to two interconnected software engineering aspects: i) application development and ii) testing.

The design framework is based on a UML model-based approach to support the formalized description of the application level use case scenarios and the fault-tolerant protocols and mechanisms developed in course of the project. Our main achievements are an application

development framework that is based on OMG's Model Driven Architecture concept defining an own UML profile for designing applications for the HIDENETS middleware platform. To achieve this, we had to harmonize the models of the underlying HIDENETS middleware services in the ad-hoc and infrastructure domains. To support the application of our approach, we have developed a domain specific editor (support for designs using our UML profile), we have elaborated design patterns (support for applications using the HIDENETS middleware services) and configuration and code generation methods (support for implementation of designs relying on our approach).

The testing framework targeted at the removal of design faults that is well suited to address the challenges and technical constraints raised by applications and services in the mobile, ad-hoc nature of the typical HIDENETS applications. The special requirements of testing mobile applications based on ad-hoc communication initiated the definition of a special, formally well-founded modelling language for describing scenarios in mobile settings. To check whether an execution trace satisfies requirements and covers test purposes, special graph matching methods had to be elaborated to test the described scenarios against the requirements.

Keyword list: application development, testing, model-based.

Table of Contents

Bibliography

[ADS] Gergely Pintér, Zoltán Micskei, András Kövi, Zoltán Égel, Imre Kocsis, Gábor Huszerl, András Pataricza: *Model-Based Approaches for Dependability in Ad-Hoc Mobile Networks and Services*, In Rogério de Lemos, Felicita Di Giandomenico, Cristina Gacek, Henry Muccini and Marlon Vieira (eds.): Architecting Dependable Systems V. (LNCS 5135) pp. 150-174., 2008

[AIS] Service Availability Forum: *Application Interface Specification* http://www.saforum.org/specification/AIS_Information/

[AP233] ISO TC 184 (Automation systems and integration) SC 4 (Industrial Data): *Industrial automation systems and integration – Product data representation and exchange, Application Protocol 233 (Systems engineering data representation)*

[AUTOSAR] AUTomotive Open System Architecture http://www.autosar.org

[Bai] F. Bai, N. Sadagopan, A. Helmy: *The IMPORTANT Framework for Analyzing the Impact of Mobility on Performance of Routing for Ad Hoc Network*, AdHoc Networks Journal - Elsevier Science, Vol. 1, Issue 4, pp. 383-403, Nov. 2003

[CavFil] A. Cavarra and J.K. Filipe: *Formalizing Liveness-Enriched Sequence Diagrams Using ASMs*, Abstract State Machines, 2004, pp. 62-77.

[D1.1] Markus Radimirsch, Erling V. Matthiesen, Gábor Huszerl, Manfred Reitenspieß, Mohamed Kaâniche, Inge Einar Svinnset, António Casimiro, Lorenzo Falai (HIDENETS Consortium): *Use case scenarios and preliminary reference model*, HIDENETS D1.1 Deliverable http://www.hidenets.aau.dk/Public+Deliverables

[D2.1] António Casimiro (editor), Andrea Bondavalli, Mário Calha, Marius Clemetsen, Alessandro Daidone, Mônica Dixit, Zoltán Égel, Lorenzo Falai, Felicita Di Giandomenico, Audun F. Hansen, Gábor Huszerl, András Kövi, Marc-Olivier Killijian, Tom Lippmann, Yaoda Liu, Erling V. Matthiesen, Henrique Moniz, Anders Nickelsen, Jimmy J. Nielsen, Thibault Renier, Matthieu Roy, José Rufino, Hans-Peter Schwefel, Inge-Einar Svinnset (HIDENETS Consortium): *Resilient Architecture (final version)*, HIDENETS D2.1.2 Deliverable, December 2007 http://www.hidenets.aau.dk/Public+Deliverables

[D2.2] António Casimiro (editor), Jesper Grønbæk, András Kovi, Anders Nickelsen, Hans Reiser, Thibault Renier, Hans-Peter Schwefel (HIDENETS Consortium): *Service level resilience solutions for the infrastructure domain*, HIDENETS D2.2 Deliverable, June 2008 http://www.hidenets.aau.dk/Public+Deliverables

[D2.3] António Casimiro (editor), Henrique Moniz, Mônica Dixit, Luis Marques, Marc-Olivier Killijian, Erling V. Matthiesen, Alessandro Daidone, Matthieu

Roy (HIDENETS Consortium): *Service level resilience solutions for the ad-hoc domain*, HIDENETS D2.3 Deliverable, June 2008
http://www.hidenets.aau.dk/Public+Deliverables

[D4.1.2] Paolo Lollini, Andrea Bondavalli (editors), Jean Arlat, Marius Clemetsen, Lorenzo Falai, Audun Fosselie Hansen, Martin Bøgstadt Hansen, Mohamed Kaâniche, Karama Kanoun, Máté Kovács, Yaoda Liu, Melinda Magyar, István Majzik, Erling V. Matthiesen, Anders Nickelsen, Jimmy J. Nielsen, Jakob Gulddahl Rasmussen, Thibault Renier, Hans-Peter Schwefel (HIDENETS Consortium): *Evaluation methodologies, techniques and tools*, HIDENETS D4.1.2 Deliverable, December 2007
http://www.hidenets.aau.dk/Public+Deliverables

[D4.2.2] Paolo Lollini, Andrea Bondavalli (editors), Francesco Brancati, Andrea Ceccarelli, Marius Clemetsen, Ludovic Courtès, Alessandro Daidone, Geir Egeland, Lorenzo Falai, Jesper Grønbæk, Ossama Hamouda, Audun Fosselie Hansen, Martin B. Hansen, Mohamed Kaâniche, Marc-Olivier Killijian, Máté Kovács, István Majzik, Erling V. Matthiesen, Leonardo Montecchi, Anders Nickelsen, Jimmy J. Nielsen, David Powell, Jakob G. Rasmussen, Thibault Renier, Hans-Peter Schwefel (HIDENETS Consortium): *Application of the evaluation framework to the complete scenario*, HIDENETS D4.2.2 Deliverable, December 2008
http://www.hidenets.aau.dk/Public+Deliverables

[D5.1] András Kövi, András Pataricza, Bálint Rákosi, Gergely Pintér, Zoltán Micskei (HIDENETS Consortium): *UML profile and design patterns library*, HIDENETS D5.1 Deliverable, March 2007
http://www.hidenets.aau.dk/Public+Deliverables

[D5.2] Hélène Waeselynck, Zoltan Micskei, Minh Duc N'Guyen, Nicolas Rivière (HIDENETS Consortium): *Preliminary testing framework and methodology*, HIDENETS D5.2 Deliverable, December 2007
http://www.hidenets.aau.dk/Public+Deliverables

[D6.2] Irene de Bruin (editor), António Casimiro, Mario Calha, Geir Egeland, Lorenzo Falai, Peter Frejek, Jesper Grønbæk, Sonia Heemstra de Groot, Audun Fosselie Hansen, Gábor Huszerl, Mohamed Kaâniche, Marc-Olivier Killijian, András Kövi, Tom Lippman, Yaoda Liu, Erling V. Matthiesen, Anders Nickelsen, Jimmy Nielsen, Gergely Pintér, Matthieu Roy, Hans-Peter Schwefel, Inge-Einar Svinnset (HIDENETS Consortium): *Specification HIDENETS laboratory set-up scenario and components*, HIDENETS D6.2 Deliverable, October 2007
http://www.hidenets.aau.dk/Public+Deliverables

[D6.3] Manfred Reitenspieß (editor), Irene de Bruin, António Casimiro, Mario Calha, Zoltan Egel, Geir Egeland, Lorenzo Falai, Bjarke Freund-Hansen, Sonia Heemstra de Groot, Audun Fosselie Hansen, Gábor Huszerl, Marc-Olivier Killijian, András Kövi, Tom Lippmann, Luis Marques, Erling V. Matthiesen, Anders Nickelsen, Gergely Pintér, Matthieu Roy, Hans-Peter Schwefel, Gaëtan Séverac, Inge-Einar Svinnset, Christophe Zanon (HIDENETS Consortium): *Experimental proof-of-concept set up HIDENETS*, HIDENETS D6.3 Deliverable, August 2008

http://www.hidenets.aau.dk/Public+Deliverables

[D6.4] Zoltan Egel (editor), Irene de Bruin, António Casimiro, Mario Calha, Geir Egeland, Lorenzo Falai, Bjarke Freund-Hansen, Sonia Heemstra de Groot, Audun Fosselie Hansen, Gábor Huszerl, Marc-Olivier Killijian, András Kövi, Tom Lippmann, Luis Marques, Erling V. Matthiesen, Anders Nickelsen, Gergely Pintér, Matthieu Roy, Hans-Peter Schwefel, Gaëtan Séverac, Inge-Einar Svinnset, Christophe Zanon, Manfred Reitenspieß (HIDENETS Consortium): *Documentation and Evaluation of the experimental work*, HIDENETS D6.4 Deliverable, December 2008
http://www.hidenets.aau.dk/Public+Deliverables

[Gue] M. K. Guennoun : *Architectures Dynamiques dans le Contexte des Applications à Base des Composants et Orientés Services*, PhD Thesis, University of Toulouse III, France, Dec. 2006

[Hal] H. H. Hallal, S. Boroday, A. Petrenko and A. Ulrich: *A formal approach to property testing in causally consistent distributed traces*, Formal Aspects of Computing, Volume 18, Issue 1 (March 2006), pp. 63 - 83.

[HaMa] David Harel and Shahar Maoz: *Assert and negate revisited: Modal semantics for UML sequence diagrams*, Software and Systems Modeling, 7(2):237–253, May, 2008

[HDoW] HIDENETS (EU Framework Programme 6 IST STREP) Project Proposal Annex I – *Description of Work*
http://rcl.dsi.unifi.it/projects/HIDENETS-DoW.pdf

[Hua] Q. Huang, C. Julien, and G. Roman: *Relying on Safe Distance to Achieve Strong Partitionable Group Membership in Ad Hoc Networks*, IEEE Transactions on Mobile Computing 3, 2 (Apr. 2004)

[Klo] J. Klose: *Live Sequence Charts: A Graphical Formalism for the Specification of Communication Behavior*, PhD thesis, C. v.O. Universitat Oldenburg (2003)

[Küs] J. Küster-Filipe: *Modelling Concurrent Interactions*, Theoretical Computer Science, 351(2):203–220, 2006
http://dx.doi.org/10.1016/j.tcs.2005.09.068

[MARTE] A UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded systems, Beta 2
http://www.omg.org/docs/ptc/08-06-08.pdf

[MDA] Object Management Group: *Model Driven Architecture Guide v.1.0.1*, 2003
http://www.omg.org/cgi-bin/doc?omg/03-06-01
http://www.omg.org/mda/

[Mics] Z. Micskei, H. Waeselynck, M. D. Nguyen, and N. Riviere: *Analysis of a group membership protocol for Ad-hoc networks*, LAAS Technical Report *no. 06797*, November 2006

[MiWae] Z. Micskei, H. Waeselynck: *A survey of UML 2.0 sequence diagrams' semantics*, LAAS Report no. 08389, August 2008

[MOF] Object Management Group: *Meta Object Facility*

http://www.omg.org/mof/

[MSC] Z. 120 ITU-T Recommendation Z. 120: *Message Sequence Chart (MSC)*, ITU-TS, Geneva, April 2004

[Ngu] M.D. Nguyen, H. Waeselynck, N. Rivière: *Testing mobile computing applications : towards a scenario language and tools*, 6th Workshop on Dynamic Analysis (WODA 2008), ACM Press, Washington D.C, USA, July 2008

[NS] S. McCanne and S. Floyd: *ns Network Simulator* http://www.isi.edu/nsnam/ns/

[OCL] Object Management Group: *UML 2.0 OCL Specification*, 2003 http://www.omg.org/docs/ptc/03-10-14.pdf

[OpenAIS] The OpenAIS Standards Based Cluster Framework http://www.openais.org

[OpenSAF] The Open Service Availability Framework http://www.opensaf.org

[QoSFT] Object Management Group: *UML Profile for Modelling QoS and FT Characteristics and Mechanisms, v1.0*, OMG Specification, 2006 http://www.omg.org/technology/documents/formal/QoS_FT.htm

[RAS] Object Management Group: *Reusable Asset Specification*, 2005 http://www.omg.org/technology/documents/formal/ras.htm

[RSA] IBM Rational Software Architect official home page, http://www.ibm.com/software/awdtools/swarchitect/websphere/

[SAC] R Sangwan, C Neill, M Bass, Z El Houda: *Integrating a software architecture-centric method into object-oriented analysis and design*, The Journal of Systems & Software, 2008

[SAF] Service Availability Forum http://www.saforum.org/home

[SPT] Object Management Group: *UML Profile for Schedulability, Performance and Time, version 1.1*, 2005 http://www.omg.org/docs/formal/05-01-02.pdf

[Stö] H. Störrle: *Trace Semantics of Interactions in UML 2.0*, Technical Report. Institut für Informatik, Ludwig-Maximilians-Universität München, 2004

[SysML] Object Management Group: *Systems Modeling Language (OMG SysML™), V1.1* http://www.sysml.org/docs/specs/OMGSysML-v1.1-AS-ptc-08-05-16.pdf

[Szat] Szatmári Zoltán: Introducing dynamism to SA Forum cluster, MSc Thesis, BME DMIS, 2008

[Urb] Urbanics Gábor: Introducing dynamism to SA Forum cluster, MSc Thesis, BME DMIS, 2008

[UML TP] Object Management Group: *UML 2.0 Testing Profile, V1.0*, July 2005

http://www.omg.org/technology/documents/formal/test_profile.htm

[UMLinf] Object Management Group: *Unified Modeling Language (OMG UML), Infrastructure, V2.1.2*
http://www.omg.org/docs/formal/07-11-04.pdf

[UMLsup] Object Management Group: *Unified Modeling Language (OMG UML), Superstructure, V2.1.2*
http://www.omg.org/docs/formal/07-11-02.pdf

[Waes] H. Waeselynck et al.: *Mobile Systems from a Validation Perspective: a Case study*, Proc. of the 6th International Symposium on Parallel and Distributed Computing (ISPDC'07), IEEE CS Press, Austria, Jul. 2007

[XMI] Object Management Group: *XML Metadata Interchange (XMI) Specification*
http://www.omg.org/docs/formal/03-05-02.pdf

Abbreviations

ADTB	Application Development Test-Bed
AIS	Application Interface Specification
AMF	Availability Management Framework
API	Application Programming Interface
AUTOSAR	AUTomotive Open System Architecture
CB	Cooperative Backup
CM	Conceptual Model
CORBA	Common Object Request Broker Architecture
COTS	Components off the Shelves
CSI	Component Service Instance
CWM	Common Warehouse Metamodel
DOM	Document Object Model
DSE	Domain Specific Editor
DTD	Document Type Definition
ECU	Electronic Control Unit
EMF	Eclipse Modeling Framework
EMOF	Essential MOF
FT	Fault Tolerance
GMP	Group Membership Protocol
GraphSeq	Graph matching tool for Sequences of configurations
GRM	General Resource Model
HA	High Availability, Highly Available
HV	Head Vehicle
JAXB	Java API for XML Binding
LSC	Live Sequence Charts
MARTE	UML Profile for Modeling and Analysis of Real-time and Embedded systems
MDA	Model Driven Architecture
MDD	Model Driven Development
MOF	Meta Object Facility
MSC	Message Sequence Charts
MSD	Modal Sequence Diagrams

OCL	Object Constraint Language
OEM	Original Equipment Manufacturers
OMG	Object Management Group
PA	Performance Analysis
PDSS	Platoon Driver Support System
PIM	Platform Independent Model
PSM	Platform Specific Model
QoS	Quality of Service(s)
RAS	Reusable Asset Management
RPC	Remote Procedure Call
RSA	Rational Software Architect
RT	Real Time
SA	Service Availability (see SAF)
SAF	Service Availability Forum, SA Forum™
SD	Sequence Diagrams
SI	Service Interface
SPT	Profile for Schedulability, Performance and Time
SU	Service Unit
SUT	System Under Test
SV	Slave Vehicle
SysML	System Modeling Language
TERMOS	TEst Requirement language for MObile Setting
TTFD	Timely Timing Failure Detection
UML	Unified Modeling Language
V&V	Verification and Validation
XMI	XML Metadata Interchange
XML	Extensible Markup Language

1 Executive Summary and Introduction

The current deliverable summarizes the results of the HIDENETS project related to extending the state of the art software engineering methods and tools in order to cope with the specific requirements of highly dependable mobile system design. This work was carried out in two cooperating tasks: (1) "UML design patterns and workflow" developing a UML based design methodology for mobile applications and (2) "Testing methodology and framework" developing methodologies to support the testing of resilient mobile applications and services. After this short common introduction of the whole work, Section 2 and 3 report on the results of the two tasks and Section 4 concludes the deliverable.

The HIDENETS project has focused on the challenges of offering highly dependable services over (and for) an inherently unreliable – ad-hoc, mobile, IP-based – network of components. Other work packages aimed at providing solutions for a resilient architecture, middleware and communication and for the quantitative evaluation of these solutions, while our work targeted the special needs of the design (development and testing) of applications running on the HIDENETS architecture in this specific environment.

Following the actual state of the art, both of our application development framework and testing framework are based on model driven methods. Both of them have defined their special extensions of the standard *Unified Modeling Language (UML)* to support modelling of application systems and worked on different models of the system.

The modelling languages have to be different because the aim of building application models differs in the two frameworks. Modelling always means abstracting, i.e. focusing only on the relevant aspects of the modelled entity. The relevant aspects are different when the goal of modelling is application development or testing. When developing applications the focus is more on what a single node perceives from the whole system, while testing focuses on a more global view. For example, mobile nodes outside of the reachability of a given node cannot be present in the application development oriented model of the given node, while they may play an import role in a testing oriented one. Then again, both modelling languages are based on the same basis (UML and the extending OMG standards), therefore they do not conflict anyway, they can be combined whenever it is required, and they can be supported by the same modelling tools.

Both tasks started with a study of the existing solutions, continued with identifying the open research points, elaborating new methods and then building prototype tools for the new methods.

The main results of the reported work are:

- A model based development approach for HIDENETS applications, its supporting tool-chain and design patterns, and a conceptual model of HIDENETS-related features (Section 2.2)
- A formal model of the interfaces of HIDENETS middleware services in the ad-hoc and infrastructure domain (Section 2.3)
- Support for the application design activities: a domain specific editor and source code and configuration generators (Section 2.4)
- A scenario language for specifying test requirements of mobile systems (Section 3.2)
- A graph matching tool to support the evaluation of test traces with respect to the defined test requirements (Section 3.3)

2 Design Framework

Two complementary main activities could be distinguished in the tasks of the HIDENETS project that are related to extending the state of the art software engineering methods and tools in order to cope with the specific requirements of highly dependable mobile system design. The first one is aimed at the development of a UML based design methodology. While other project tasks aim at the improvement of the services of the HIDENETS middleware, this one focuses on the development of applications using this middleware, that is, on the effective and efficient utilization of the other project results. When speaking about application development we consider applications that take advantage of those services rather than implementing mobility and dependability related features for themselves.

This section summarizes the project achievements in the field of application development: First we describe what could be utilized from the existing design methodologies and standards, and what middleware concepts and solutions are provided by the other tasks of the project (a short and high-level description of the HIDENETS middleware architecture), see Section 2.1. Then based on them we outline our modelling approach, the heart of our model based application design framework in Section 2.2. As the HIDENETS applications have to rely on the HIDENETS middleware services, our application design framework has to incorporate a model of the concepts of that middleware. The third subsection (Section 2.3) documents this HIDENETS metamodel. In Subsection 2.4 we introduce our results for providing support for the application designers such as specific editors for modelling and specific automatic generators for utilizing the well-known automation potential of model based design methodologies in implementing and deploying HIDENETS applications. For an overview how we have proved the applicability of our approach by completing a prototype application development project, see Section 2.5. Preliminary versions of this framework were reported in the HIDENETS project deliverable [D5.1] and in the book chapter [ADS].

When a (group of) application designer(s) decides to build a distributed application for a run-time environment that is corresponding to the one targeted by the HIDENETS concept (highly dependable IP-based networks and services), he may choose our application development approach:

- We support building application designs with a specific UML model based language that includes model element types for HIDENETS specific concepts. The domain specific editor is a tool to ease the work with this special modelling language.
- HIDENETS design patterns are reusable solutions to commonly occurring problems in the application design in HIDENETS environment. Most of these patterns are related to the application of the different HIDENETS middleware services.
- When the application design is already documented in details, there are some standard steps that can be easily automated. Code generators and configuration generators take charge of some non-creative tasks of translating the models into code fragments or configuration descriptors. The manual execution of these tasks is usually highly inefficient and error prone, but it can only be avoided if there is tool support for the given run-time environment. That is why HIDENETS prefers standard solutions wherever it is possible.

2.1 Goal of the Framework

The goal of our application design framework is to provide support for the application developer who has to efficiently design highly dependable distributed applications running on several nodes both in the ad-hoc and infrastructure domains, where the nodes are connected by inherently

unreliable IP-based networks. We started our work by studying the standard design methodologies, the application domain specific standards and specifications, and the HIDENETS middleware architecture that the applications have to rely on.

2.1.1 Design Methodologies

From the very beginning we aimed at technical solutions where "Resilience and availability of services deployed either in an ad-hoc domain or on dedicated servers in the Internet, have to be taken into account on a system design level, since the components are inherently unreliable."(from "HIDENETS – Description of Work" [HDoW])

We have worked on a design methodology that can reach this goal. We have chosen a modelling based approach since modern design methodologies fit under the model-driven architecture (MDA) [MDA] initiative in which applications are primarily designed and specified by their (semi-)formal model. MDA and UML have been the glue to interlock our efforts in supporting both the application development and testing.

Other modern design methodologies are based on a software architecture centric view [SAC]. This approach is well known for primarily focusing on the quality attributes of the target system and effectively supporting the development in dominantly distributed scenarios. However, the main drawback is the omission of the finer grade design of the system components, wherefore we decided rather for a model driven approach.

2.1.2 Standards and Specifications

In order to facilitate re-use of previously published field expertise, conformance to the existing standards was a main objective of our project. Thus emphasis was put on the integration of our work to corresponding widely known and industrially accepted conceptual frameworks such as

- *Unified Modeling Language* (UML) 2.0 [UMLsup, UMLinf] as a general purpose modelling language, for serving as basis notation for several specialized modelling languages
- *Reusable Asset Specifications* (RAS) [RAS] for supporting reusability through consistent, standard structuring and packaging
- *Systems Modeling Language* (SysML) [SysML] as the industry standard for modelling complex software-intensive systems, widely applied for modelling in systems engineering
- *Automotive Open System Architecture* (AUTOSAR) [AUTOSAR] as standardized automotive software architecture for modelling automotive industry specific artifacts
- OMG's *UML Profile for Schedulability, Performance and Time* (SPT) [SPT] and its (to be accepted) successor OMG's *UML Profile for Modeling and Analysis of Real-time and Embedded systems* (MARTE) [MARTE] for modelling application–platform interaction
- OMG's *UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics* (2006) (QoS and FT Profile) [QoSFT] for deriving QoS related concepts
- SAForum's *Application Interface Specification* (AIS) [AIS] integrating the most important means for HA assurance

into a unifying framework.

A primary objective in our work was the clarification of the relation between the HIDENETS specific model-driven design and implementation approach and the standards in the field. The main

policy was to assure compliance to the major standards as far as possible in order to support the future reuse of development technologies in the mainstream of embedded applications. In this context, SysML was taken into account as one of the evolving standards. An important domain-specific standard package related to the target pilot field of application (automotive) is AUTOSAR where the applicability of the AUTOSAR concepts in the HIDENETS project has already been analyzed, but future compliance checks are needed as this package still undergoes rapid evaluations. Our project focuses on the vehicle-to-vehicle or vehicle-to-roadside communication, while AUTOSAR actually concentrates only on intra-vehicle (mainly E/E and control) systems, while promising more in the software domain for the future. By the AUTOSAR currently covered intra-vehicle components are usually hidden from the HIDENETS application developers, in this aspect they only serve the implementation of the execution platform.

These last standards in the above list were used to derive general dependability and time concepts, e.g. they clearly appear in the data type definitions of our HIDENETS specific profiles. Fundamental time-related concepts are similar in SPT and MARTE. Our data types actually are derived from the already accepted SPT standard, but they can be easily fitted to the new profile once MARTE will be officially released. The middleware service interfaces offered to the applications are aligned wherever possible with SA Forum interfaces (e.g. AIS). Because of the significant functional differences between the relevant services in the ad-hoc and infrastructure domain, it was dominantly possible in case of the services in the latter one. One of the lessons learned in this project is that the integration to well-known standards is possible, even if working in a very specific field like the one of HIDENETS, and that it enables efficient knowledge re-use.

In this deliverable we only give a short introduction of two of the studied standards and specifications, for more details please read the HIDENETS deliverable D5.1. For the sake of shortness Appendix A only discusses the two most important standards: OMG's UML and the SA Forum interface specifications, because a basic knowledge of them is required in the further sections.

The application of the standard UML profiles is not at all specific to the HIDENETS project. We only elaborated special solutions for modelling HIDENETS specific aspects where there are no standard ones; otherwise we prefer the application of existing standards. This way we have only defined UML profiles which target aspects of the system under development that are orthogonal to the aspects covered by the here introduced standards. We have designed our profiles to avoid any conflicts with the existing standards. Therefore they can be simultaneously applied in a single model without any modification.

2.1.3 HIDENETS Architecture

One of the main objectives of the HIDENETS project is "*to define a resilient architecture and to develop a range of middleware solutions (i.e. algorithms, protocols, services) for resilience to be applied in the design of highly available, reliable, and trustworthy networking solutions*" (from the original project proposal). Other project deliverables (e.g. [D2.2] and [D2.3]) describe this architecture in details. The HIDENETS architecture defines communication and middleware services and provides them for the applications through programming interfaces. The application developer has to study the provided functionalities and interfaces to be able to develop applications on top of this HIDENETS middleware. Sections 2.3.2 and 2.3.3 describe the interface and usage of some of these services (defined for nodes operating in the ad-hoc and infrastructure domains, respectively) in more details.

The architecture consists of the following services for direct access of the application components running on the nodes in the ad-hoc domain (other *oracle* and *communication* services have not to be used by the applications directly, therefore they are omitted here):

- Diagnostic and Reconfiguration Manager
- QoS Coverage Manager
- Replication Manager
- Proximity Map
- Cooperative Data Backup
- Intrusion-Tolerant Agreement

However, not all of them are directly accessible from the application layer.

Because of the differences in the requirements a different approach was taken for the server nodes in the infrastructure domain. Since there are standard solutions that can satisfy the requirements of a HIDENETS node in this domain, we suggest that a SA Forum compliant middleware will be used there (this standard was shortly introduced in Section 2.1.2, for a more detailed description see Appendix A).

In the following, we suppose that the applications have to run on HIDENETS nodes and therefore they can rely on the services provided by the HIDENETS architecture.

2.2 Overview of the Modelling Activities

This section presents an *overview on the modelling activities* carried out in the HIDENETS project focusing on the refined design framework. The section is built up of three subsections: (i) first we outline the *application of MDA principles* in the organization of the work (Section 2.2.1) then (ii) *identify the key tasks* to be carried out for achieving our goals (Section 2.2.2), and finally (iii) briefly summarize the *key features of the tool-chain* developed by us for supporting model-driven development in HIDENETS (Section 2.2.3).

2.2.1 Model Driven Architecture in the Context of the HIDENETS project

This subsection outlines the *application of MDA principles* in the organization of modelling activities carried out in the framework of the HIDENETS project.

OMG's *Model Driven Architecture* (MDA) initiative aims at organizing the process of model transformations and code synthesis into a well-structured framework. MDA considers two abstraction levels: (i) the level of *meta-models* (i.e., meta-models of modelling or programming languages) and (ii) the level of actual *models* (i.e., software models or source code). For the two abstraction levels there are three key steps of the process: (i) platform independent modelling, (ii) platform specific modelling and (iii) implementation.

- In the *platform independent modelling* (PIM) step engineers prepare an early model of the system without taking into consideration the restrictions and benefits of the target platforms (possibly chosen in the future). This step allows the modellers to focus only on the actual task, re-use design patterns without being heavily influenced by platform specific features. Platform independent models are usually constructed in the pure UML language without any platform-specific extensions.

- During the *platform specific modelling* (PSM) step the PIM model is mapped to the resources available on the actual target platform. PSM models are also prepared in UML but with the application of a target specific modelling profile. UML profiles provide further specialization of UML's built-in general concepts (e.g., device) according to the needs of the target platform (e.g., the control domain may need to distinguish devices as sensors and actuators).
- MDA's final step is *implementation* that according to the OMG's proposal should be carried out by automatic code synthesis as much as possible. This automatic code synthesis step transforms the PSM model to source code of the application. Although currently available code synthesis solutions are still unable to fully eliminate the need for manual programming, they have already been reported as beneficial solutions for significantly increasing productivity and software quality.

Our modelling work in the context of the HIDENETS project was inspired and organized according to the MDA initiative. Below we briefly highlight the MDA-related aspects of our work.

As outlined above the preparation of an application's *platform independent model* (PIM) does not need much customization of the UML base language thus we expect application developers preparing software for the HIDENETS platform to use UML as-is, in the PIM phase.

Our solutions enter the picture at the *platform specific modelling* (PSM) step: application developers should be empowered with such HIDENETS-specific modelling artefacts that represent the resources that are available for applications running on either the infrastructure or the ad-hoc part of the HIDENETS platform. These language extensions are obviously prepared as UML profiles (see Section 2.3.2 and 2.3.3 for an introduction to these profiles and the corresponding metamodels).

With respect to the *implementation* step we provide (i) automatic source code generators that are able to synthesize some key infrastructure-related parts of applications and (ii) configuration file generators for the automatic construction of configuration descriptors for SA Forum/AIS middleware implementations (see Section 2.2.3 for an overview of the tool-chain and Section 2.4 for further details on individual utilities).

2.2.2 Key Phases of Modelling Activities

According to the MDA organization outlined above we have to enable software modellers to indicate HIDENETS-specific features in the PSM step. Below we indicate those steps that are needed for constructing the *HIDENETS profile for UML,* providing a *domain specific editor* as design support tool and communicating best practices in the form of *design patterns*.

In practice this means that for each group of HIDENETS-related application features we have to (i) establish a *conceptual model*, (ii) construct a *UML profile* according to the conceptual model, (iii) present some *application examples* of the profile in order to support its easy understanding and widespread application, finally (iv) we have to provide *design patterns* to enable re-use of best practices and successful software organization recipes:

- Constructing a *conceptual model* of a HIDENETS-related feature (e.g., replication of critical components on the infrastructure side) actually means the *collection of key concepts* (e.g., service groups, service units, components, checkpoints, replication schemes, etc.) and indicating their *association relations* (e.g., a service group contains the description of the replication scheme) and *packaging hierarchy* (e.g., a service group as a package contains any number of service units etc.). We present conceptual models as ordinary UML *class diagrams* where key concepts appear as classes and their relations as associations,

inheritance, packaging etc. At this point we are only aiming at the understanding of these features without actually connecting these concepts to built-in UML artefacts.

- Having constructed the conceptual model and unambiguously discussed the meaning of features appearing there, we are ready to *connect them to built-in UML artefacts*. This step is necessary for profile construction since we have to indicate the relation of newly introduced (HIDENETS-related) concepts to original UML artefacts. In this step
  - The *classes* identified in the conceptual modelling step will be represented by newly introduced *metaclasses.* For example, components whose state can be saved into a checkpoint are to be represented by a new metaclass *Replicable Component* that is derived from the built-in UML *Component* metaclass.
  - Associations will be represented by instances of the built-in UML *Association* (or *Association Class*) metaclass. For example, if service groups are represented by the metaclass *Service Group* and replication schemes are represented by the metaclass *Replication Scheme*, indicating the containment relation between service groups and replication schemes is possible through an *Association* metaclass instance connecting the two artefacts through *Property* features. (The metaclass *Service* Group is derived from the built-in UML *Component* metaclass, and the metaclass *Replication Scheme* is derived from the built-in UML *Class* metaclass)
  - and packaging hierarchy can be indicated similarly.

  The metamodel extension built this way provides the foundations of the corresponding UML profile for HIDENETS-related applications, since we only have to
  - assign stereotypes to newly introduced metaclasses,
  - define some necessary tagged values and
  - save the profile in a format suitable for the actual modelling environment

  (see Sec. 2.3.1 for further details on meta-modelling, profiles etc.).
- Even a well-documented profile's application requires some understanding of the target platform and considerable expertise in software modelling. In order to achieve the hoped-for wide acceptance of the HIDENETS profile we should present some *modelling examples* showing the application of the profile in practice. Examples are presented in this document as HIDENETS-related fragments of software models shown in static structure diagrams (class, package, component, etc. diagrams) where HIDENETS-related artefacts are indicated by the stereotypes introduced in the previous step.
- Finally in case of some complex or even error-prone HIDENETS-related features it may be beneficial to indicate the *best practices* of application organization. This knowledge is delivered by the document as a set of *design patterns* that are collections of various model fragments involving both static structure and dynamic diagrams (e.g., class, package, component, statechart, interaction diagrams, etc.).

We will follow this four-step organization (conceptual modelling, metamodelling and profile construction, application examples and design patterns) for presenting the modelling activities in Section 2.3.

2.2.3 An Overview on the Tool-Chain

Having organized our modelling and implementation efforts according to the MDA initiative there is a straightforward way for the implementation of our design/modelling support tools into a coherent *tool chain* whose components focus on the modelling, implementation and deployment steps of application development (see Figure 1):

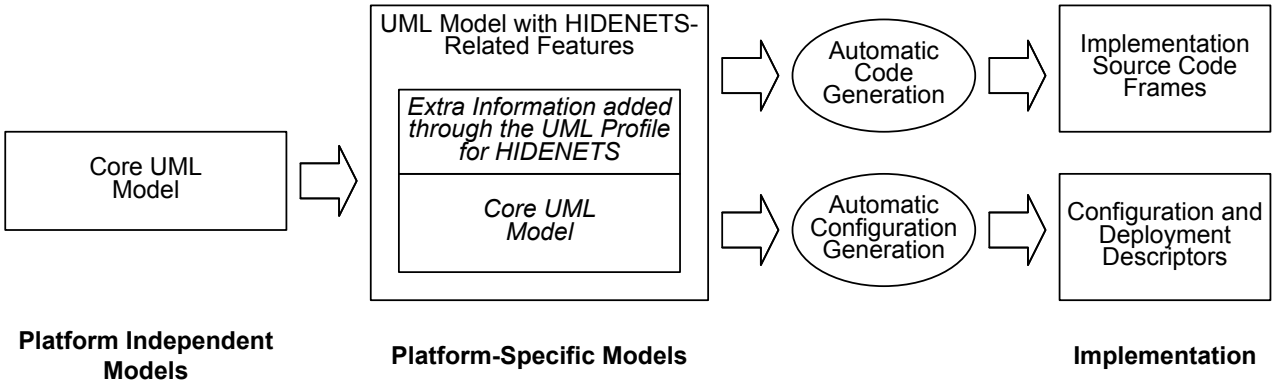


Figure 1: **Key Elements of the Tool-Chain**

- *Modelling HIDENETS-related features of application* is obviously based on the application of the UML profile for HIDENETS as outlined above. Although a model stereotyped according to the HIDENETS profile carries all the information needed for automatic processing, users may find beneficial a slightly easier to remember notation than pure stereotypes and tagged values. In order to achieve this, our profile is extended with a user-friendly visual notation (i.e., special icons assigned to HIDENETS-specific features). We call this extended profile the *Domain Specific Editor (DSE) for the HIDENETS platform*. The DSE is implemented as an extended profile to be used in the IBM Rational Software Architect modelling environment (for more information on the modelling environment, see [RSA]). Note that the DSE does not add conceptually new features to the concepts introduced above: The model saved by the DSE is an ordinary UML model stereotyped according to the HIDENETS profile. The key contribution of the DSE is to present this model in a more user-friendly way that would be done barely using a modelling tool and the plain profile. See Sec. 2.4.1 for further details about the DSE.
- The implementation of some key infrastructure-related features is based on the Service Availability Forum's (SA Forum) middleware (see App. A). Since application development for the SA Forum middleware enforces a *specific application organization* it is beneficial to support the programmers by automatically synthesizing the necessary code structure on the basis of the application model (obviously annotated according to the HIDENETS profile). Sec. 2.4.2 presents an overview on our *automatic code generator* that is capable of processing an application model and synthesizing the necessary code frames (i.e., declarations of methods, data structures, etc.) similarly to usual code wizards found in modern integrated development environments.
- Another key task related to the SA Forum-based infrastructure implementation is the actual deployment of application components to the distributed fault-tolerant computing resource structure. The configuration of this middleware is by far not trivial and necessitates considerable expertise in the field. Our *configuration file synthesis solution* discussed in Sec. 2.4.2 aims at substituting this labour intensive and error-prone task by automatically

generating configuration and deployment descriptors based on the application's model (obviously annotated according to the HIDENETS profile).

To put together: our achievements are not only logically organized according to the MDA initiative but their actual implementation also reflects this approach. Due to using UML, a standard modelling language with a clearly defined profile, tool providers are free to develop further components for the tool chain.

2.3 Modelling HIDENETS Related Application Features

After a short introduction to metamodelling and profile construction this section presents the discussion of HIDENETS-related application modelling artefacts according to the four-step organization scheme introduced above (i.e., conceptual modelling, metamodelling and profile construction, application examples and design patterns).

2.3.1 Introduction on Metamodelling and Profile Construction

UML is a complex modelling language providing support for various modelling activities from early specification phases until the planning of software deployment. Due to its wide application area, the UML metamodel is built up of several hundred metaclasses resulting in a very complex structure whose modification and extension requires a considerable expertise in metamodelling. In order to enable the easy and straightforward extension of UML, the concept of *profiles* was introduced into the language. A UML profile is a lightweight extension of the language i.e., profiles are targeted for adding extra platform-specific features to the language without inherently changing the structure of built-in metaclasses.

A profile may specify *new metaclasses* that are derived from built-in concepts. Since new metaclasses are derived from already existing ones, the visualization of new artefacts in a modelling environment does not need much effort either: when having to add a specialized element into the model, the user has to insert the original (built-in) element and indicate that in this case she/he is not referring to the plain built-in concept but some other metaclass derived from it. This indication of instances of newly introduced metaclasses is carried out by applying a *stereotype* to the metaclass instance in concern. Stereotypes are textual strings within guillemots (e.g., <<stereotype>>). Theoretically there is no one-to-one correspondence between newly introduced metaclasses and stereotypes, but it is a good practice to introduce one stereotype for each new metaclass and use the same name for them (e.g., *Sensor* metaclass and <<sensor>> stereotype). The modelling environment may enable the user to assign icons to stereotypes that may even replace the built-in graphical symbol of the original metaclass.

As newly introduced metaclasses may have *attributes*, the modeller has to be able to assign values to these attributes; this value assignment is carried out by *tagged values*. A tagged value specification is a key-value pair whose key is the name of the attribute. Modelling environments typically enable the user to specify these value assignments in a tabular format or a property view.

Profiles are extensively used for adapting UML to a *platform-specific modelling task* e.g., an embedded system engineer may need to explicitly indicate sensor and actuator devices in a deployment diagram but UML does not provide built-in concepts for this. Thus a UML profile for embedded systems may introduce two new metaclasses: *Sensor* and *Actuator* from the built-in *Device* metaclass and introduce the corresponding stereotypes <<sensor>> and <<actuator>> possibly with easy to recognize icons assigned to them. It may be important to indicate the latency of a sensor thus the *Sensor* metaclass can have an attribute *latency* and the modeller can specify the latency of sensors in the model by a tagged value.

2.3.2 Ad-hoc Domain

This sub-section discusses our modelling activities corresponding to HIDENETS-related application features focusing on the *ad-hoc domain* by presenting the *conceptual model*, *metamodelling* and profile construction, *application example* and *design patterns*. For easiest understanding we chose here the cooperative backup and timely timing failure detection services as examples, which are quite easy to understand for non HIDENETS experts. Conceptual model, metamodel and design patterns were also developed for the remaining services and can be found in the corresponding detailed UML model (see the prototype part of this deliverable).

2.3.2.1 Cooperative Backup

Conceptual Model

The conceptual model of an application's view about the cooperative backup feature involves two key classes: (i) the *CB_Client* client (i.e., the actual application using the cooperative backup feature) and (ii) the *CB_Storage* storage (i.e., the reliable storage facility where backup data is saved to).

The class diagram in Figure 2 shows the key idea behind the conceptual model. *Clients* of a cooperative backup scenario are ordinary classes (components, etc.) that access the storage in a relatively raw form, i.e., the storage is seen as a stream that can accept a sequence of bytes. The data stored previously can be retrieved from the *storage* again as a byte sequence.

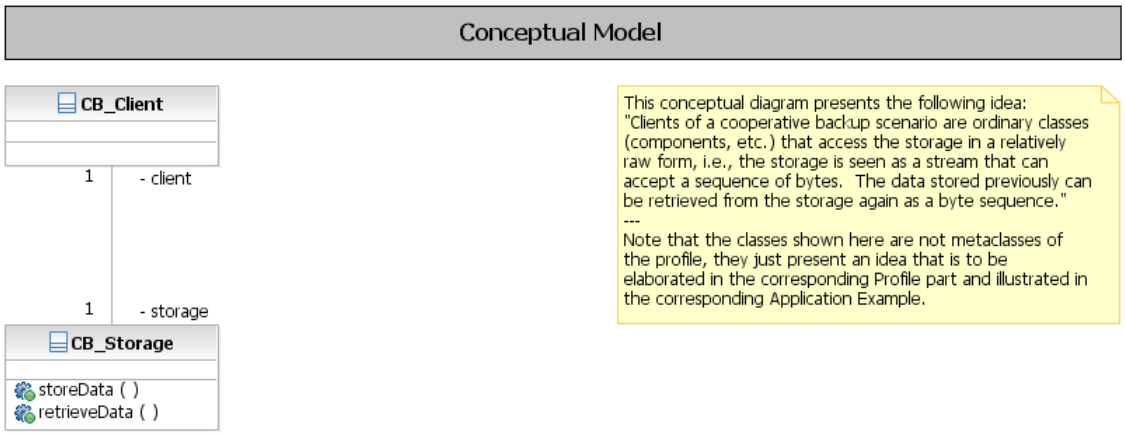


Figure 2 : Conceptual Model of Cooperative Backup Activities

Metamodelling and Profile Construction

Having outlined the idea in the quite informal class diagram above, we have to assign *metaclasses* to the concepts introduced above and *derive* these new metaclasses from built-in UML artefacts. The newly introduced metaclasses and their relations to core UML features are shown in Figure 3.

Storage and client are represented by metaclasses *CBStorage* and *CBClient* respectively; both derived from the UML *Class* metaclass.

There are two new operation kinds (derived from the UML *Operation* metaclass) *CBStoreOperation* and *CBRetrieveOperation* corresponding to storing and retrieving backup data in/from the storage respectively. These operations belong to the storage (metaclass *CBStorage*).

Storage and clients are connected through a specialized association *CBAssociation* (derived from the UML built-in *Association* metaclass). As associations connect classes through properties, we also introduced metaclasses *CBClientProperty* and *CBStorageProperty* derived from the UML

built-in *Property* metaclass and representing the client from the point of view of the storage and vice versa respectively.

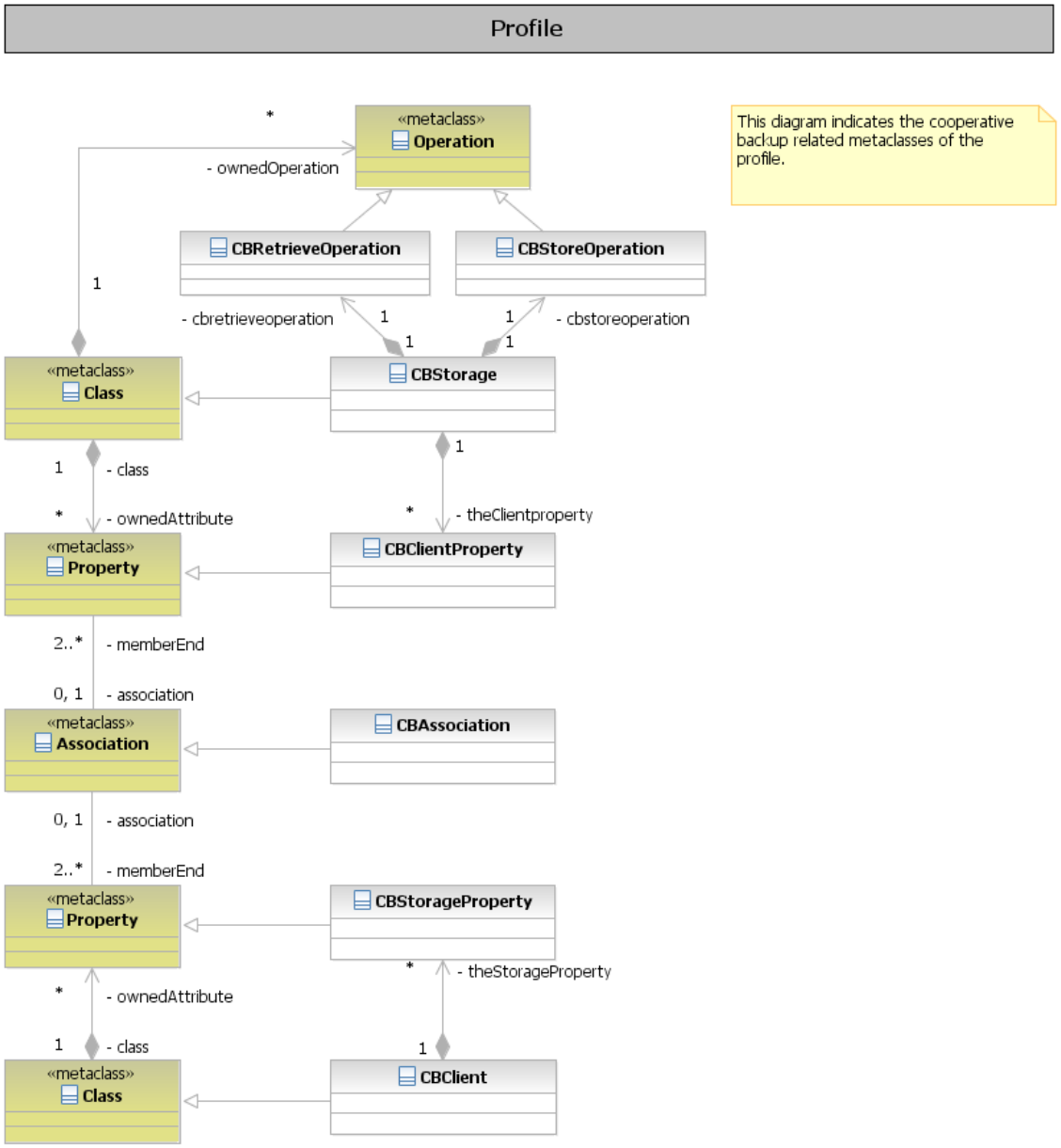


Figure 3 : Metamodelling and Profile Construction for Cooperative Backup Activities

The brief description of newly introduced metaclasses is as follows:

- *CBRetrieveOperation:* An operation provided by the storage facility in a cooperative backup scenario used for retrieving previously saved data.
- *CBStoreOperation:* An operation provided by the storage facility in a cooperative backup scenario used for saving data.
- *CBStorage:* Cooperative backup storage. Instances of this metaclass represent stable storage facilities used for cooperative backup. The actual implementation of the storage may be application field specific e.g., flash memory, disk space etc.

- *CBClientProperty:* A property owned by a cooperative backup storage indicating the client(s) associated to the storage.
- *CBAssociation:* Cooperative backup association. Instances of this association metaclass indicate relations between client and storage facilities in a cooperative backup scenario.
- *CBStorageProperty:* A property owned by a cooperative backup client indicating the storage(s) associated to the client.
- *CBClient:* Cooperative backup client class. Instances of this metaclass take part in a coopertive backup scenario, i.e., their state can be saved into a stable store.

The newly introduced metaclasses are directly mapped to stereotypes in the HIDENETS profile (the name of the stereotype exactly equals to the name of the corresponding metaclass):

- *CBRetrieveOperation:* Applicable to operations of classes that represent storage facilities. Semantics of the stereotype: this operation can be used for retrieving data previously saved into the storage facility.
- *CBStoreOperation:* Applicable to operations of classes that represent storage facilities. Semantics of the stereotype: this operation can be used for storing data in the reliable storage.
- *CBStorage:* Applicable to classes. Semantics of the stereotype: this class is a cooperative backup storage facility providing operations for storing and retrieving data.
- *CBClientProperty:* Applicable to properties (association roles). Semantics of the stereotype: this end of the association is a client class.
- *CBAssociation:* Applicable to associations. Semantics of the stereotype: this association connects storage facilities and clients.
- *CBStorageProperty:* Applicable to properties (association roles). Semantics of the stereotype: this end of the association is a cooperative storage facility.
- *CBClient:* Applicable to classes. Semantics of the stereotype: this class is a client of a cooperative backup storage facility.

Application Example

Figure 4 presents an example for the application of the profile fragment introduced above in the following case:

- There is a critical component in the application called *SomeCriticalComponent*.
- The component needs to be able to quickly restore its previously saved state even in case of a hardware failure thus its state is periodically saved into a stable storage.
- The storage is implemented using hard disks, the storage facility is represented by the class *SomeDiskStorage* that provides two operations *saveToDisk* and *loadFromDisk* for storing and retrieving data respectively.

It is easy to see that the client class *(SomeCriticalComponent)* is stereotyped as *CBClient* while the storage facility *(SomeDiskStorage)* is stereotyped as *CBStorage*. The association between them is stereotyped *CBAssociation* whose ends are stereotyped as *CBClientProperty* and *CBStorageProperty* respectively. Store and retrieve operations *saveToDisk* and *loadFromDisk* are stereotyped as *CBStoreOperation* and *CBRetrieveOperation* respectively.

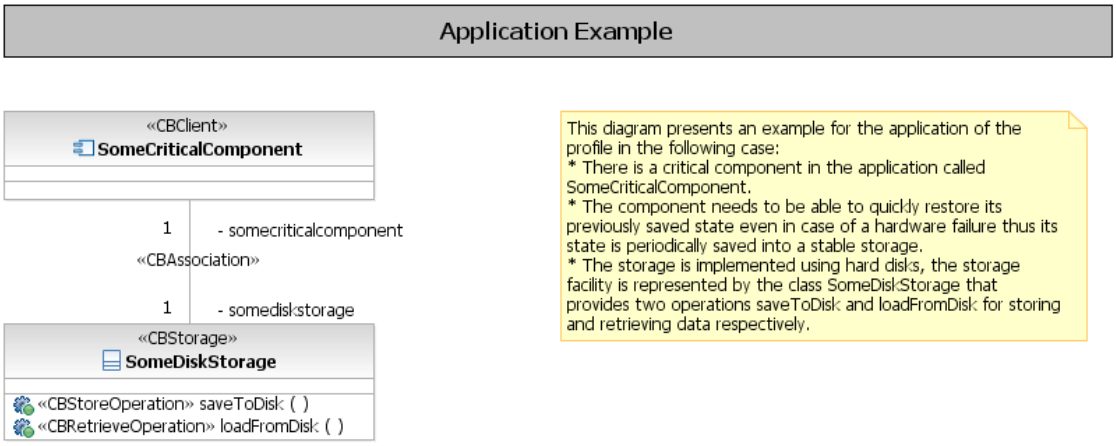


Figure 4 : Application Example for Cooperative Backup Activities

Design Pattern

The actual application of the cooperative backup facility is quite straightforward, thus the corresponding design pattern can be discussed in some easy to understand diagrams as shown below.

The design pattern can be textually formalized as follows: “If some classes need cooperative backup facility, then (i) indicate this requirement by the *CBClient* stereotype, (ii) indicate storage facilities by the *CBStorage* stereotype, (iii) highlight the relation of clients and storage facilities by adding the *CBAssociation* stereotype to their association, (iv) indicate store and retrieve operations within the storage facility by stereotypes *CBStoreOperation* and *CBRetrieveOperation* respectively and (v) periodically store the state of the critical component and restore it if necessary as shown in sequence diagrams.” (Figure 5).

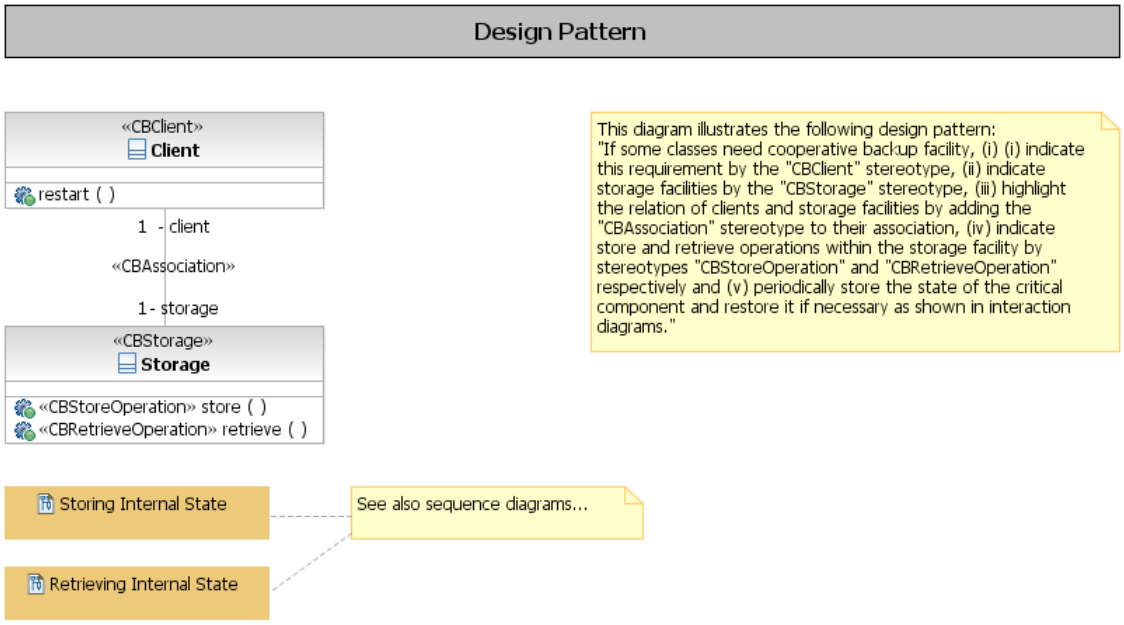


Figure 5 : Design Pattern for Cooperative Backup Activities (Class Diagram)

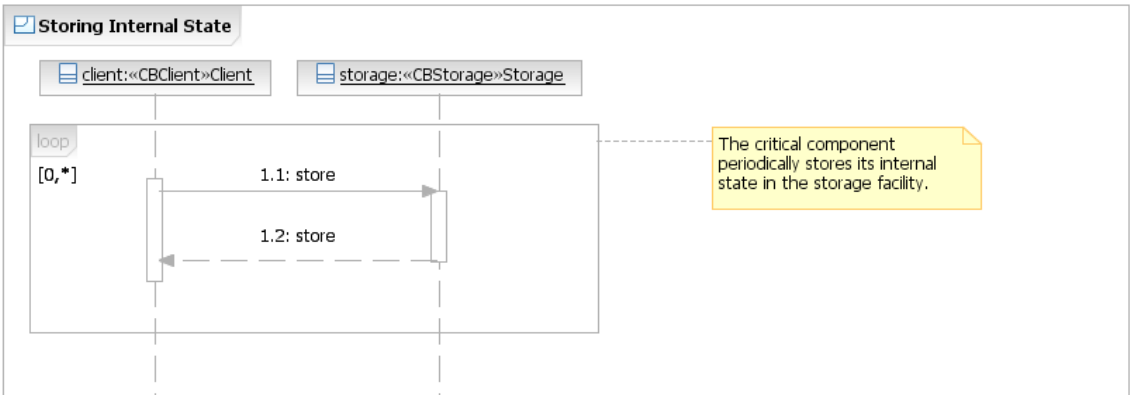


Figure 6 : Design Pattern for Cooperative Backup Activities (Storing Internal State)

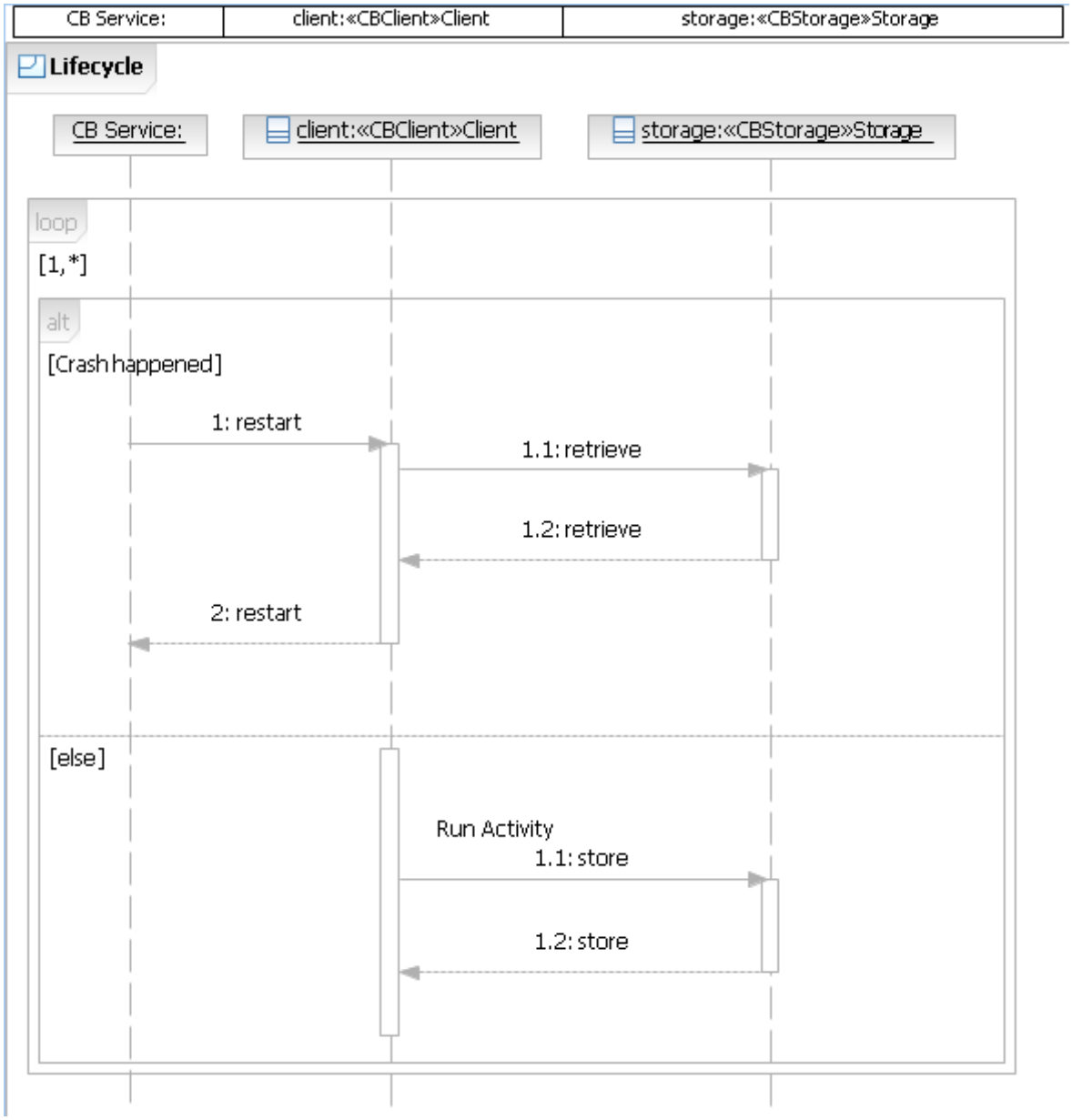


Figure 7 : Design Pattern for Cooperative Backup Activities (Retrieving Internal State)

As shown in Figure 6 the client class should periodically call the storage operation store and upon a crash or a restart the previously saved state should be restored by calling the retrieve operation as shown in Figure 7.

2.3.2.2 Timing Failure Detection

Conceptual Model

The conceptual model of an application's view about the timing failure detection feature involves five key classes: (i) a real-time service provider *(CM_RealTimeServiceProvider)* and (ii) its client *(CM_RealTimeServiceClient)*, (iii) a real-time service offered by the provider *(CM_RealTimeService)*, (iv) a timing failure notification operation in the client *(CM_TimingFailureNotificationOperation)* and (v) the real-time service agreement between the provider and the client *(CM_RealTimeServiceAgreement)* (Figure 8).

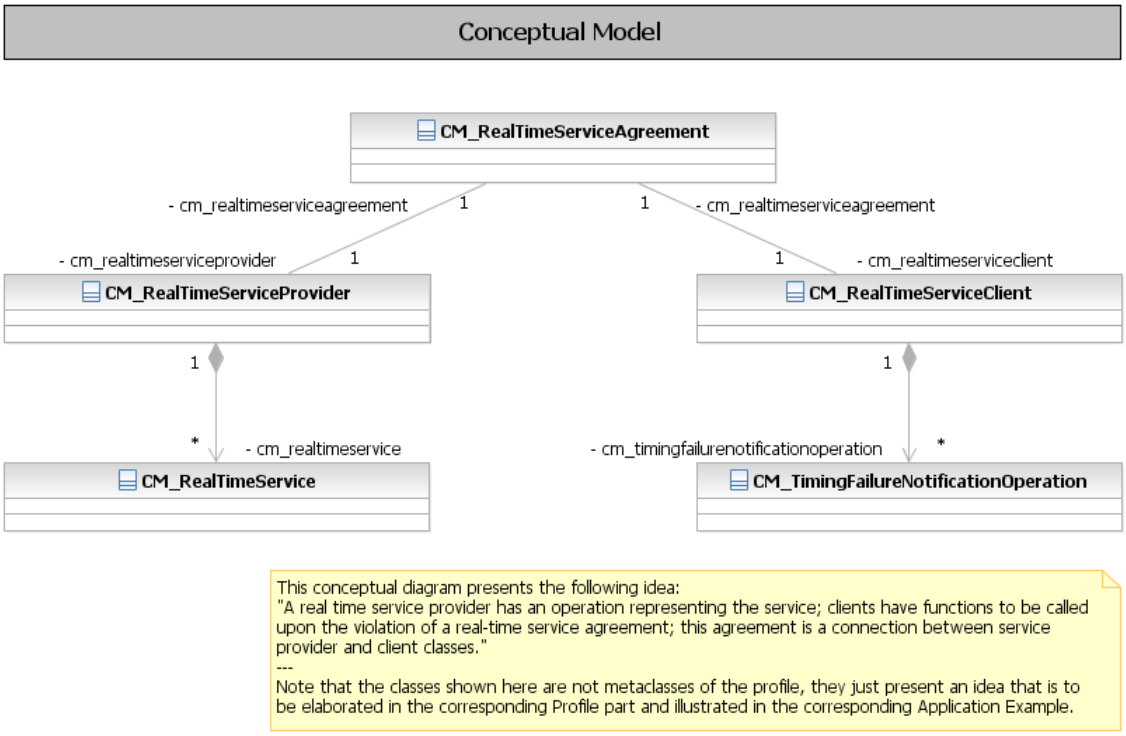


Figure 8 : Conceptual Model of Timing Failure Detection

Metamodelling and Profile Construction

Having outlined the idea of timing failure detection, we have to assign *metaclasses* to the concepts introduced above and *derive* these new metaclasses from built-in UML artefacts. The newly introduced metaclasses and their relations to core UML features are shown in Figure 9.

Real time service providers are represented by the *RealTimeServiceProvider* metaclass; its clients are represented by the *RealTimeServiceClient* metaclass, both derived from the built-in UML *Class* concept. The actual real-time service is represented by instances of the *RealTimeService* metaclass, the timing failure notification operation is mapped to the *TimingFailureNotificationOperation* metaclass both derived from the core UML *Operation* concept. As the agreement between the real-time service provider and the client is a kind of association with some properties attached the *RealTimeServiceAgreement* metaclass is derived from the UML *AssociationClass* concept. As shown in the figure the original organization of built-in metaclasses enables the seamless expression of the containment relations between our newly introduced metaclasses, e.g., since *Operation* is

contained by *Class* (through the *ownedOperation* role) we do not have to introduce new associations between our metaclasses.

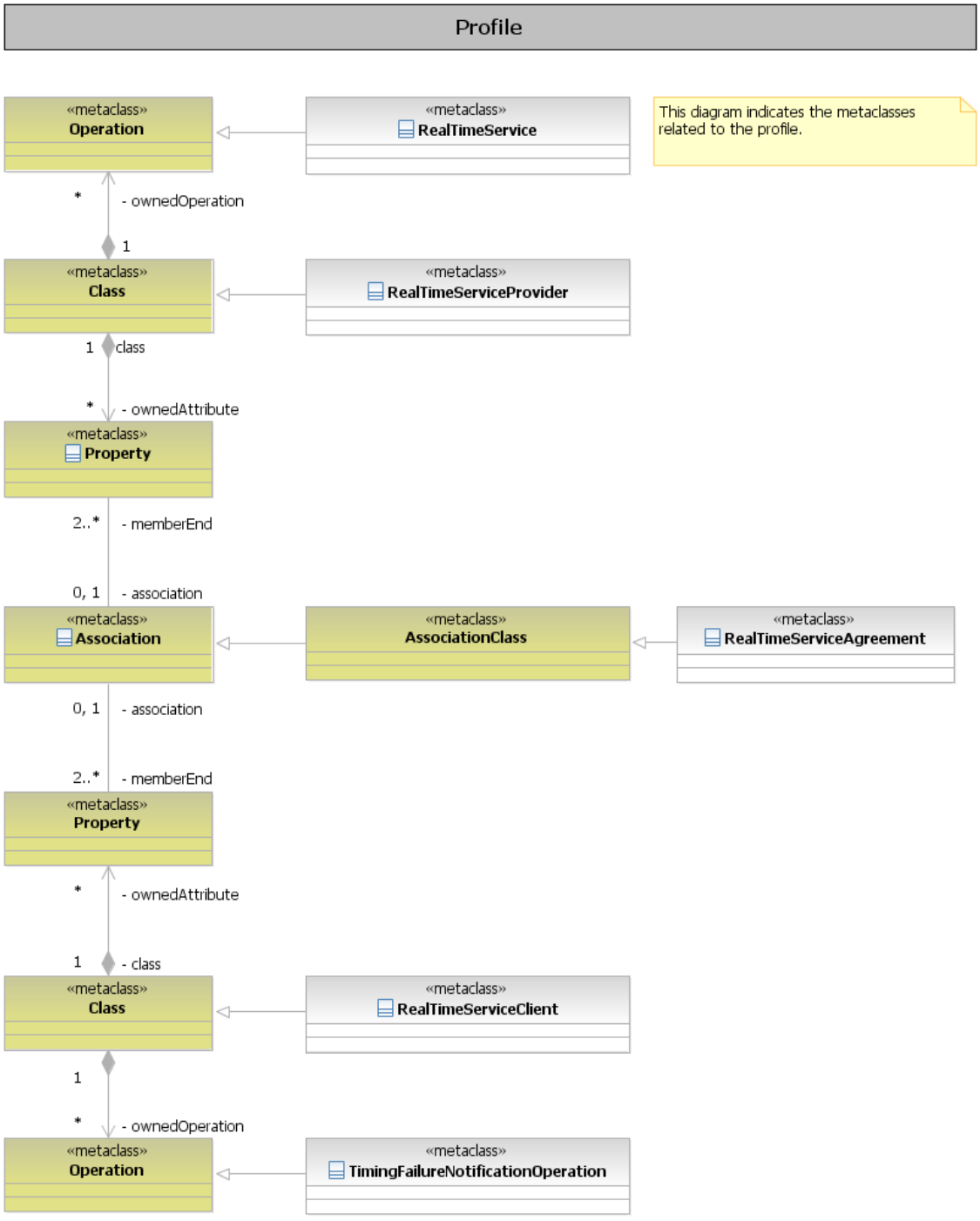


Figure 9 : Metamodelling and Profile Construction for Timing Failure Detection

The brief description of newly introduced metaclasses is as follows:

- *RealTimeService:* This metaclass represents a real-time service.
- *RealTimeServiceProvider:* This metaclass represents a provider of a real-time service.
- *RealTimeServiceAgreement:* This metaclass represents an agreement between a provider and a client of real-time services.

- *RealTimeServiceClient:* This metaclass represents the client of a real-time service.
- *TimingFailureNotificationOperation:* This metaclass represents a method of a client of a real-time service; this method is to be invoked upon the violation of the real-time service agreement enabling the client to take countermeasures.

The newly introduced metaclasses are directly mapped to stereotypes in the HIDENETS profile (the name of the stereotype exactly equals to the name of the corresponding metaclass):

- *RealTimeService:* Applicable to operations of classes that represent real-time service providers. Semantics of the stereotype: this operation delivers a real-time service.
- *RealTimeServiceProvider:* Applicable to classes. Semantics of this stereotype: this class represents a provider of a real-time service.
- *RealTimeServiceAgreement:* Applicable to association classes. Semantics of this stereotype: this association represents the agreement between a real-time service provider and its client; attributes of the association class correspond to details of the agreement.
- *RealTimeServiceClient:* Applicable to classes. Semantics of this stereotype: this class represents the client of a real-time service.
- *TimingFailureNotificationOperation:* Applicable to operations of classes that represent clients of a real-time service. Semantics of the stereotype: this operation is to be invoked upon the violation of the real-time service agreement.

Application Example

Figure 10 presents an example for the application of the profile fragment introduced above in the following case:

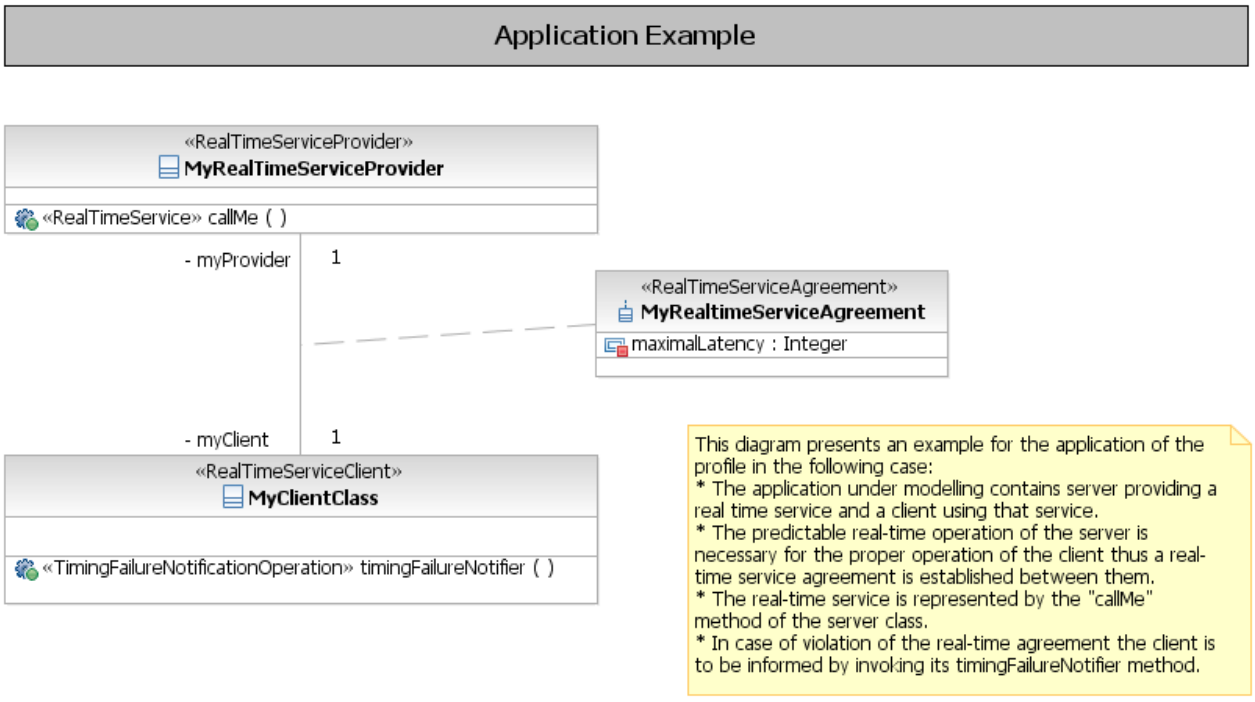


Figure 10 : Application Example for Timing Failure Detection

- There is a real-time service provider in the system called *MyRealTimeServiceProvider*. The service delivered by it is called *callMe*.
- In our case the real-time service provider has a single client called *MyClientClass*.
- The agreement between *MyRealTimeServiceProvider* and *MyClientClass* is represented by *MyRealtimeServiceAgreement*. This agreement specifies a single real-time requirement, the maximal latency (actually an integer number).
- Upon violation of the real-time service the client would like to be notified through its *timingFailureNotifier* method.

It is easy to see that MyRealTimeServiceProvider, MyClientClass and MyRealtimeService-Agreement are to be stereotyped as RealTimeServiceProvider, RealTimeServiceClient and RealTimeServiceAgreement respectively. The callMe function is the actual real-time service thus it is stereotyped as RealTimeService, while timingFailureNotifier is to be marked as TimingFailureNotificationOperation. Finally the MyRealtimeServiceAgreement association class representing the real-time agreement is obviously stereotyped as RealTimeServiceAgreement.

Design Pattern

The detailed explanation of the timing failure detection service's intended usage is shown in Figure 11 and Figure 12.

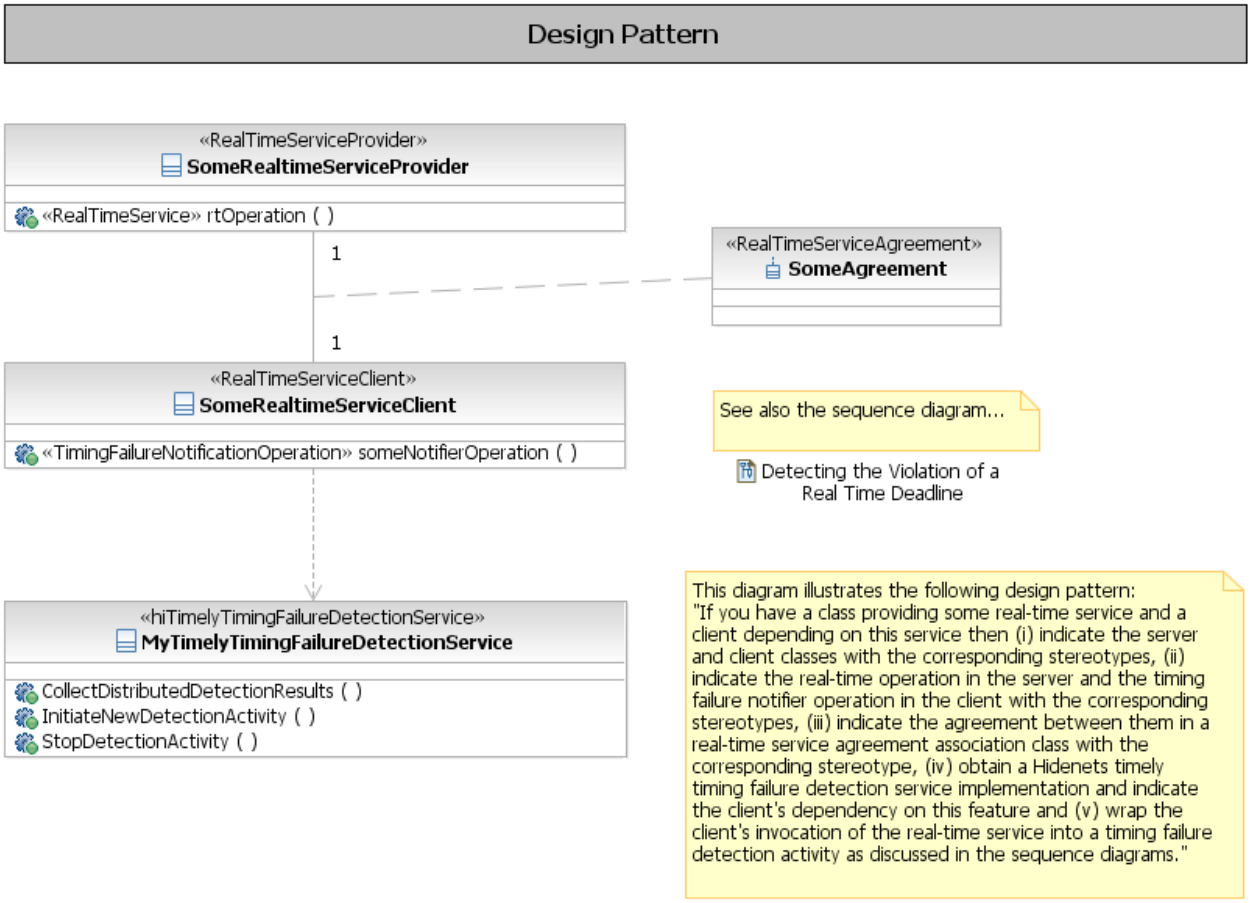


Figure 11 : Design Pattern for Timing Failure Detection (Class Diagram)

The static part of the design pattern (shown in the class diagram of Figure 11) can be textually explained as follows: "If you have a class providing some real-time service and a client depending on this service then (i) indicate the *server* and *client classes* with the corresponding stereotypes, (ii) indicate the *real-time operation* in the server and the *timing failure notification operation* in the client with the corresponding stereotypes, (iii) indicate the *agreement* between them in a real-time service agreement association class with the corresponding stereotype and (iv) obtain a HIDENETS timely timing failure detection service *implementation* and indicate the client's *dependency* on this feature."

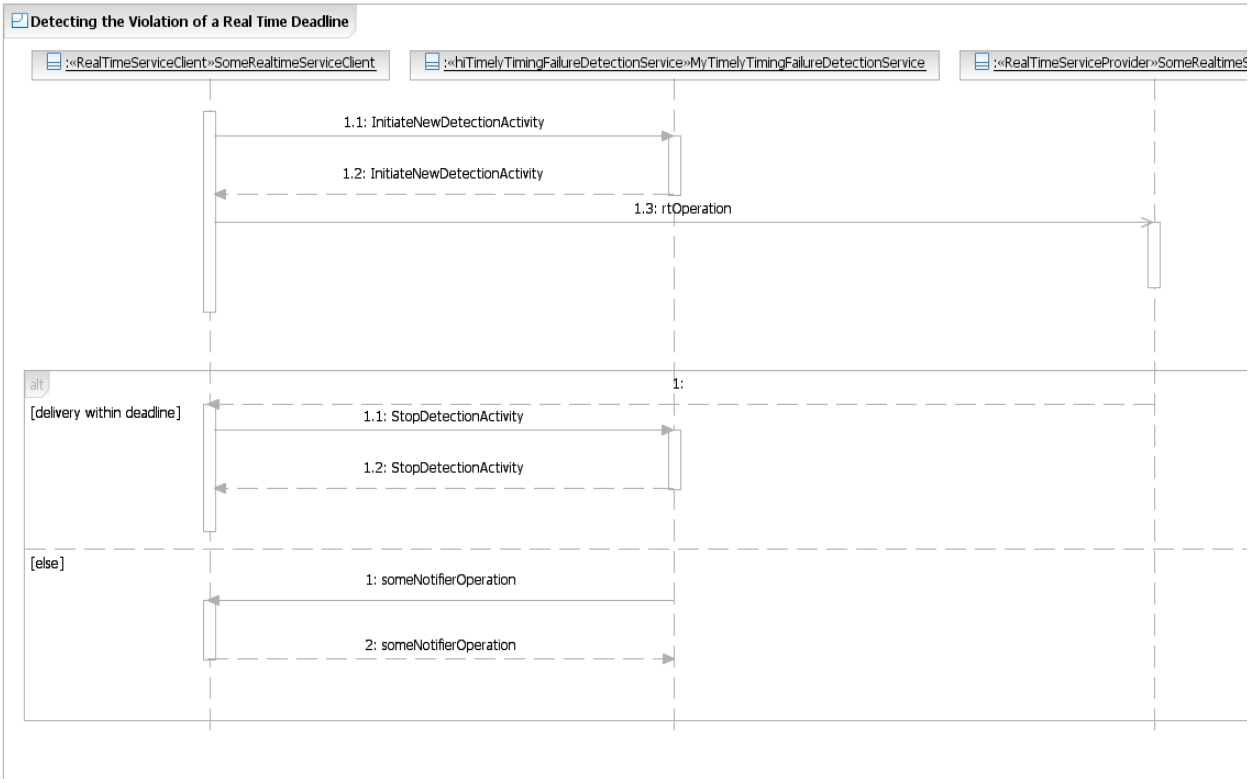


Figure 12 : Design Pattern for Timing Failure Detection (Sequence Diagram)

With respect to the dynamic part shown in Figure 12 we can say that: "(i) Before calling a real-time service initiate a detection activity, (ii) then call the service. If (iii) the server delivers the response within the agreed interval shut down the detection activity, (iv) otherwise the client class will be informed about the time-out event by the HIDENETS timing failure detection service."

2.3.3 Infrastructure Domain

Unlike in the ad-hoc domain, there are well established frameworks for providing highly available services in the infrastructure domain. For the HIDENETS architecture the *Service Availability Forum*'s *(SA Forum) Application Interface Specification (AIS)* was selected as the basis which was modelled and integrated with the services in the ad-hoc domain. This document does not aim to give a thorough description of the AIS services but focuses on the modelling framework that was developed in HIDENETS for those.

This sub-section discusses our modelling activities corresponding to HIDENETS-related application features focusing on the *infrastructure domain* by presenting the *conceptual model*, *metamodelling* and profile construction, *application example* and *structural design templates* for the Availability Management Framework and the Checkpoint service.

2.3.3.1 The Availability Management Framework (AMF)

In an AIS specifications-based system the Availability Management Framework is the entity that coordinates and monitors the services and resources in order to minimize service outages and provide fault tolerance.

Conceptual Model

In Figure 13 the simplified conceptual model of AMF is depicted. We have built this conceptual model to have a basis for our application development methodology for applications with some of their components in the infrastructure domain, as well. The model is simplified because the AMF conceptual model is more granular, but it basically extends the model described here. (The detailed conceptual model can be found in the prototype attached to this deliverable.) The base of the model is the application (represented by the *Application* class). The application comprises the services it provides and the various service providers that actually provide the service.

The applications are in fact designed to provide different types of services. Thus the Application contains several *ServiceType*s. These *ServiceTypes* describe what attributes the services have and define the default value of configuration attributes that are set by the designer in the system configuration. The *Service*s are always instances of specific *ServiceTypes*. These services are under the supervision of AMF which is responsible for monitoring their health status and controlling the system to maximize their availability.

Lastly, the *Service*s are assigned to *RedundantServiceProviders*, which are also under the control of AMF. A service provider can take either the active or the standby role on behalf of a service, and depending on its capabilities, it can take more than one assignment at a time.

The application itself is provided by a *Cluster* which is built up from *ClusterNodes* and the cluster nodes host the *RedundantServiceProviders*. AMF is aware of the cluster and the different cluster nodes but it has no control over those. The cluster management is the responsibility of the Cluster Membership service and AMF can only do administrative operations like locking and restarting on existing nodes but it is not able to add or remove nodes.

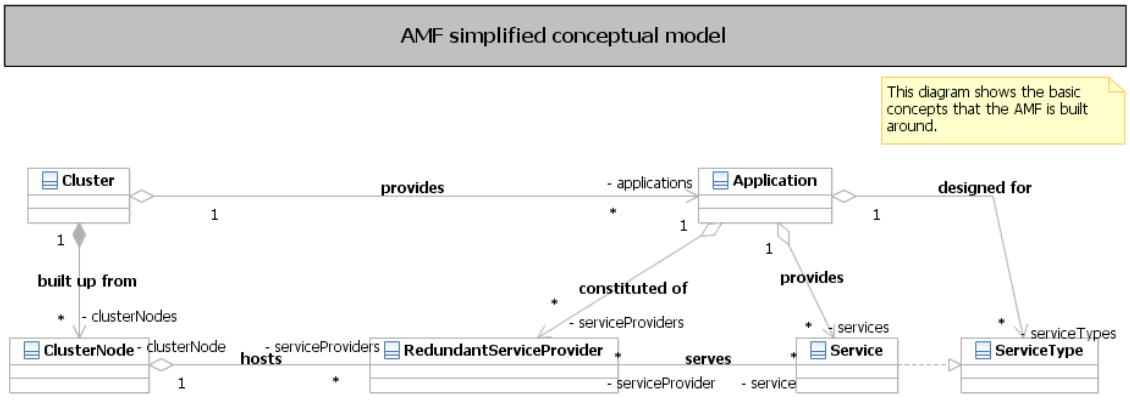


Figure 13 : Simplified Conceptual Architecture of AMF

The described model is the basis of the AMF conceptual model shown in Figure 14. The mapping of the different concepts is the following:

- *Application* is represented by *SaAmfApplication*,
- *Service* is represented by *SaAmfSI* (Service Instance,)
- *RedundantServiceProvider* is represented by *SaAmfSU* (Service Unit,)

- *ServiceType* is represented by *SaAmfServiceType*; however, it is not visualized in the figure,
- *Cluster* is represented by *SaAmfCluster*,
- *ClusterNode* is represented by *SaAmfNode*.

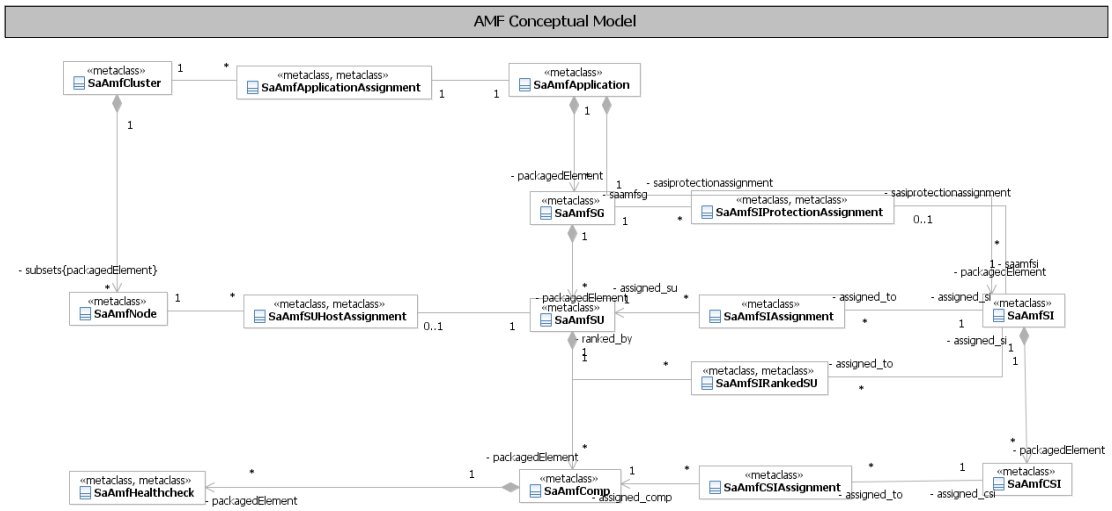


Figure 14 : AMF Conceptual Model

In addition to the simplified model, there are extensions to enable more granule management and representation of the system. These refinements are the following:

- *SaAmfSU* aggregates *SaAmfComps*. This refinement aims to enable the creation of simple components that provide basic services, and the complex services are constituted of these basic ones. Correspondingly, the *SaAmfSI* aggregates *SaAmfCSI*s (Component Service Instance), which represent the basic workload that are assigned to components.
- There can be any number of *SaAmfHealthchecks* assigned to the *SaAmfComponent*. The AMF uses these health checks to monitor the health state of the component.
- The *SaAmfSG* (Service Group) is the redundancy manager of *SaAmfSU*s and it is what protects/responsible for provision of *SaAmfSI*s. The redundancy management is done according to the policy defined as the redundancy model.

Further extensions provide ability to represent and configure assignments between different entities:

- *SaAmfApplicationAssignment* is used to configure the relationship between the cluster and the application.
- *SaAmfSUHostAssignment* is used to represent the hosting relationship between the cluster node and the service unit.
- *SaAmfSIProtectionAssignment* is used to represent the protection relationship between the service group and the service instance.
- *SaAmfSIAssignment* is used to represent the assignment relationship between the service unit and the service instance.
- *SaAmfSIRankedSU* is used to represent the preference relationship between the service instance and the service units. If a service instance is more preferred to be assigned to a service unit then that given service unit will have a higher ranking than other, less preferred ones.

- *SaAmfCSIAssignment* is used to represent the assignment relationship between the component and the component service instance.

Profile Construction

Since the AMF conceptual model is quite complex, the AMF UML profile is significantly bigger than the ones previously described in preceding sections. The complete description would exceed the size constraints of this document. However, there are two basic principles we followed during its creation:

- Entities are modelled as UML *Components*,
- Relationships are association classes if they express many-to-many relations or simple associations,
- Aggregation between entities is described as containment/packaging relation (see Figure 15.).

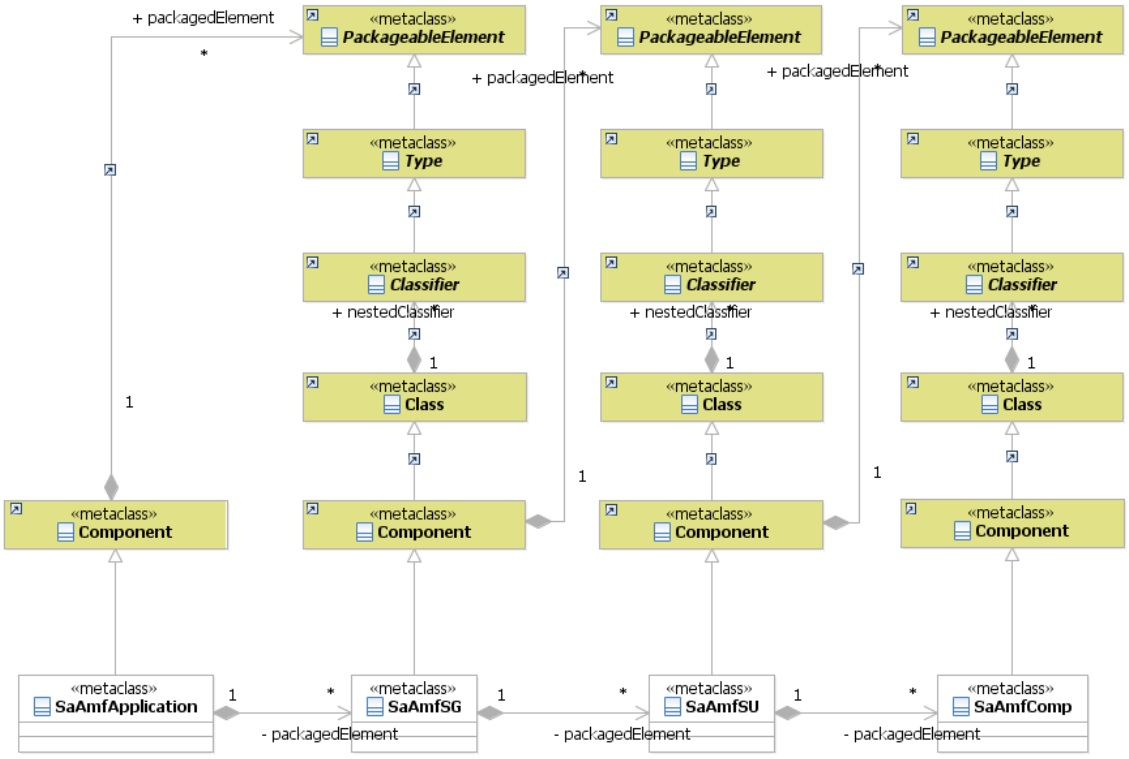


Figure 15 : AMF UML Profile Excerpt Showing the Packaging Relation between the Application, Service Group, Service Unit and Component Entities

Application Example, Structural Design Templates and Design Patterns

As AMF is a generic, multipurpose framework, design patterns for its usage can range from structural level to component implementation. In this document, structural design templates are described in detail and other patterns are mentioned only since those are more application specific and have to be adapted to the application domain.

Structural design templates are different from design patterns in the sense that they can be used as repository entries to ease the creation of configuration models, but they describe only structural

relations and, for example, service usage information cannot be assigned to them. In the description of the design templates, application examples of the AMF UML profile will be shown.

The 2N design template (see Figure 16) is used to set up a service group with the 2N redundancy model. This means that there are (at least) two service units in the service group. One of them will get only active, while the other will get only standby assignments. The input parameters for the template are:

- the name of the service group,
- the naming scheme of the service units (the final names are generated automatically),
- the *ServiceUnitType* which is the template for the actual service units (describes the components in the service unit),
- naming scheme for service instances,
- the *ServiceType* which is the template for the service instances,
- number of service instances.

It is important to note here that the services based on the *ServiceType* have to allow to be assigned to the service units based on the *ServiceUnitType*. In reality, the *ServiceType* has to contain a subset of the functionality provided by the *ServiceUnitType*.

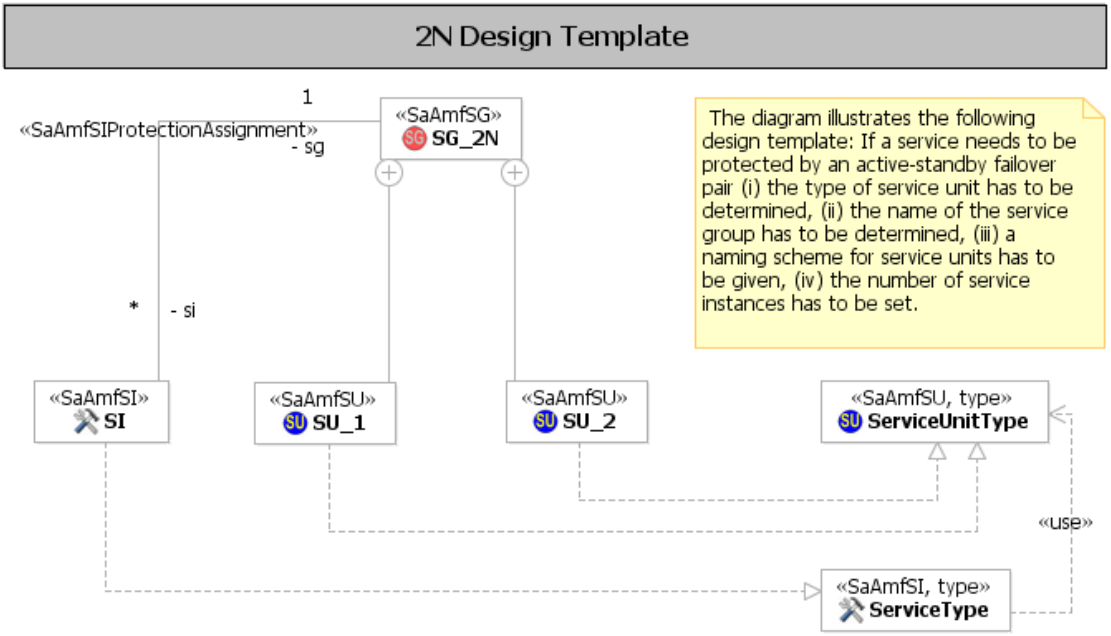


Figure 16 : 2N Design Template

The N+M design template (see Figure 17) is a generalization of the 2N design template where the number of active service units is N and the number of standby service units is M. Correspondingly, the input parameters of this design template are:

- the name of the service group,
- the naming scheme of the service units (the final names are generated automatically),
- the *ServiceUnitType* which is the template for the actual service units (describes the components in the service unit),

- number of active service units,
- number of standby service units,
- naming scheme for service instances,
- the *ServiceType* which is the template for the service instances,
- number of service instances.

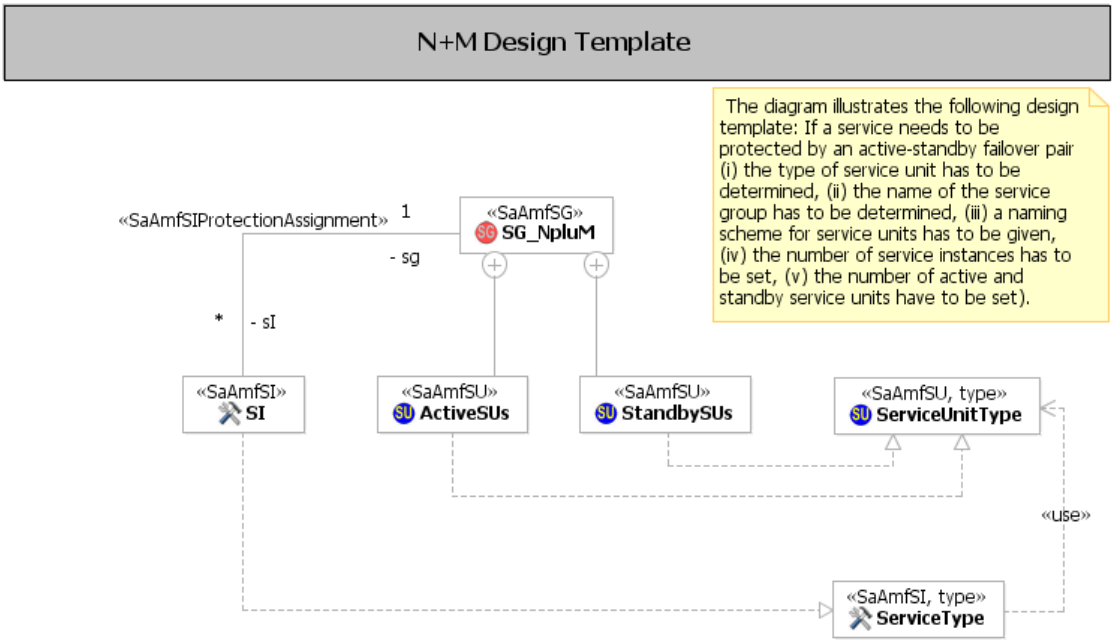


Figure 17 : N+M Design Template

Accordingly, several design templates can be created for the *N-Way*, *N-Way Active*, *No Redundancy* redundancy models, too.

Design patterns have also been created for AMF. However, as it was mentioned before, these are in most cases very application specific. It is also worth-while to note that most elements of the AMF model are only logical elements and no code or executable belongs to them. The only entity that is code and what provides the services is the *Component*. Any design patterns that concern business functionality apply only to these elements. General design patterns for *Component*s are for example:

- Threading
  - *Single threaded component*. The component is implemented in a way that all assigned Component Service Instances are running in one common thread.
  - *Multi threaded component*. The component is implemented in a way that all assigned Component Service Instances are running in separate threads.
  - *Multi process component*. Instead of starting new threads, new processes are spawned for each CSI.
- Main thread – worker thread communication
  - *Socket based communication*. Operating system sockets (socket pairs) are used for communication.

- *Shared memory based communication*. Shared memory areas are used for communication.
- *Message queue based communication*. Operating system message queues are used for communication.

Besides these, there are several other patterns which are concerned with check pointing, health checking or other application specific management functionalities.

2.3.3.2 Checkpoint Service

AMF monitors and manages the high availability state of components and applications. Stateless services can be integrated with AMF without any further modifications; however, stateful services need a way to store and restore their internal state. The Checkpoint service provides checkpoints for this purpose.

Conceptual Model

The conceptual model of the checkpoint service is depicted in Figure 18. The Checkpoint service is used by a *CheckpointClient* which can be an AMF component or any other component. Using the interfaces provided by the Checkpoint service, the *CheckpointClient* is able to create, delete, write, read *Checkpoint*s. For each client, a node local *CheckpointReplica* is created[1]. This means if two clients running on the same cluster node open the same checkpoint, then they will use the same replica. The checkpoint service synchronizes the different replicas and ensures data consistency. All checkpoint replicas are equal but one, the *active replica*. The *active replica* is used for read and write operations and all the other replicas are just data stores, the data is propagated into those, until one of them becomes the active one. There is at most one *active replica* at a time.

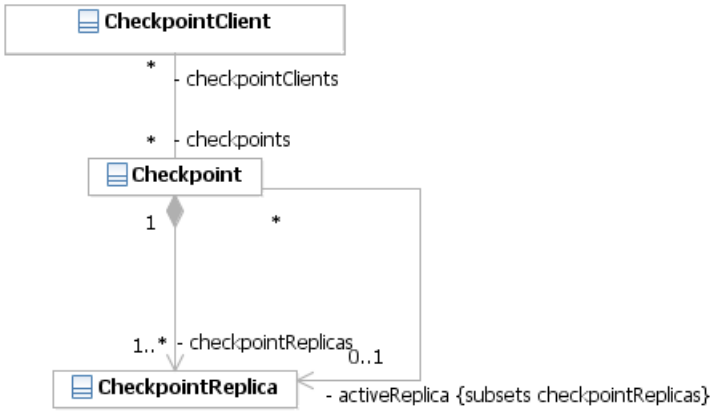


Figure 18 : Conceptual Model of Checkpoint Service

Metamodelling and Profile Construction

Based on the above overview of the entities and concepts of the checkpoint service, the metamodel and the profile are created for the service. The metamodel contains *metaclass*es and the profile describes how these are derived from the basic UML *metaclass*es. The newly created metaclasses are directly mapped to stereotypes which can be applied to specific UML entities.

The checkpoint client is represented by the *CkptClient* metaclass, while checkpoint is modelled as the *SaCkpt* metaclass. The fact that a client uses a checkpoint is manifested by the *CkptUse*

[1] Only if the „*collocated*" property of the checkpoint is set. If the checkpoint is not co-located, the checkpoint replica may reside on a different cluster node.

metaclass. Figure 19 shows the metamodel and its mapping to the basic UML concepts. The newly created metaclasses and the profile are described shortly in the following.

- *CkptClient* metaclass. Applies to the *Component* UML metaclass. Represents a client to the checkpoint service.
- *CkptUse* metaclass. Applies to the *Association* UML metaclass. Expresses the usage relation between the client and a checkpoint.
- *SaCkpt* metaclass. Applies to the *Component* UML metaclass. Represents a checkpoint.
- *SaCkptReplica* metaclass. Applies to the *Component* UML metaclass. Represents a checkpoint replica.

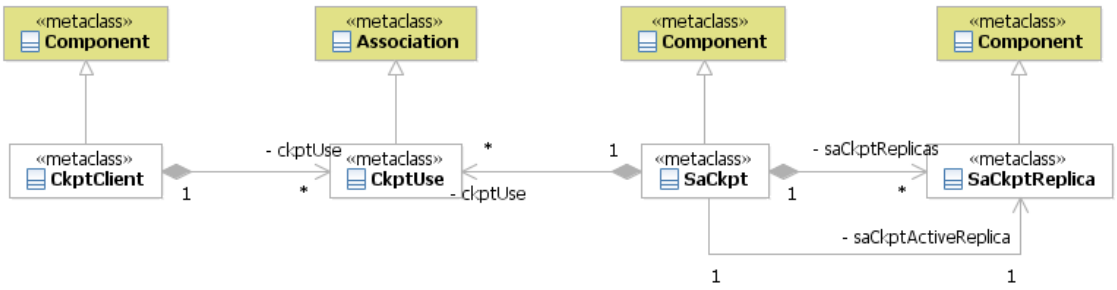


Figure 19 : Checkpoint Service Profile

Application Example

Figure 20 shows an example for the application of the checkpoint profile elements. (This and following figures use some extra icons to represent domain specific entities. The introduction of extra visual notation helps the modeller in working with these entities. For more details see Section 2.4.1.)

- There is an AMF component, called *SomeComponent,* a checkpoint client, which uses *CheckpointA* to save its internal state regularly.
- The component and the checkpoint are connected with a *CkptUse* association.

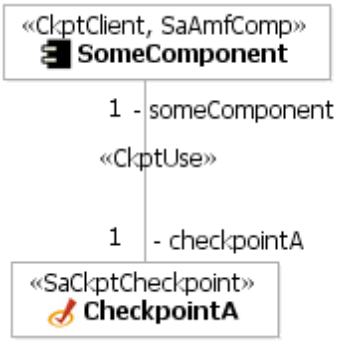


Figure 20 : Checkpoint Entities Application Example

Design Pattern

An intended use of the checkpoint service may be when an AMF component is stateful and it has to maintain its internal state despite component errors. Figure 21 shows the intended use of the checkpoint service in this case.

The AMF component (*SomeComponent*) has three different operations:

- *loadCheckpoint()* is used for reading up the contents of the checkpoint and restoring the saved state,
- *saveCheckpoint()* is used for saving the internal state into the checkpoint,
- *stateUpdatedNotification()* is used when a hot standby component is created that has to keep its internal state in synch with the active component.

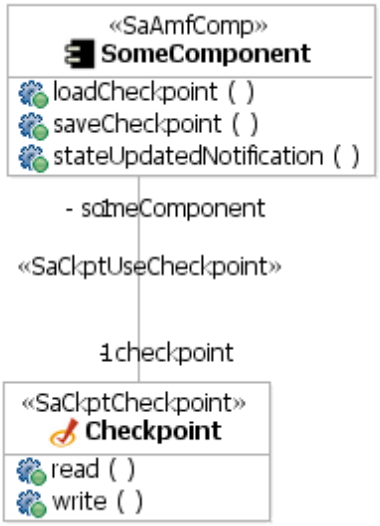


Figure 21 : Checkpointed Component Design Pattern

The basic use cases for the component are shown in Figure 22 and Figure 23. The first one shows the case when the component is started, and it checks if there is any saved state that should be restored, while Figure 23 shows what happens during the failover.

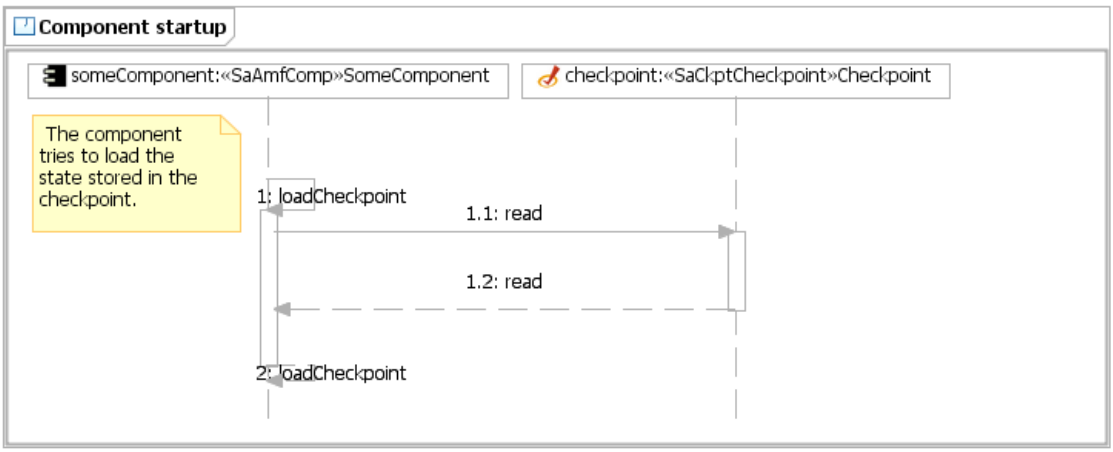


Figure 22 : Checkpointed Component Startup

At failover, the component that previously had the standby assignment is assigned the active role by the AMF. This process is called the failover. During the failover, the component reads up the state data from the checkpoint and restores it as the internal component state.

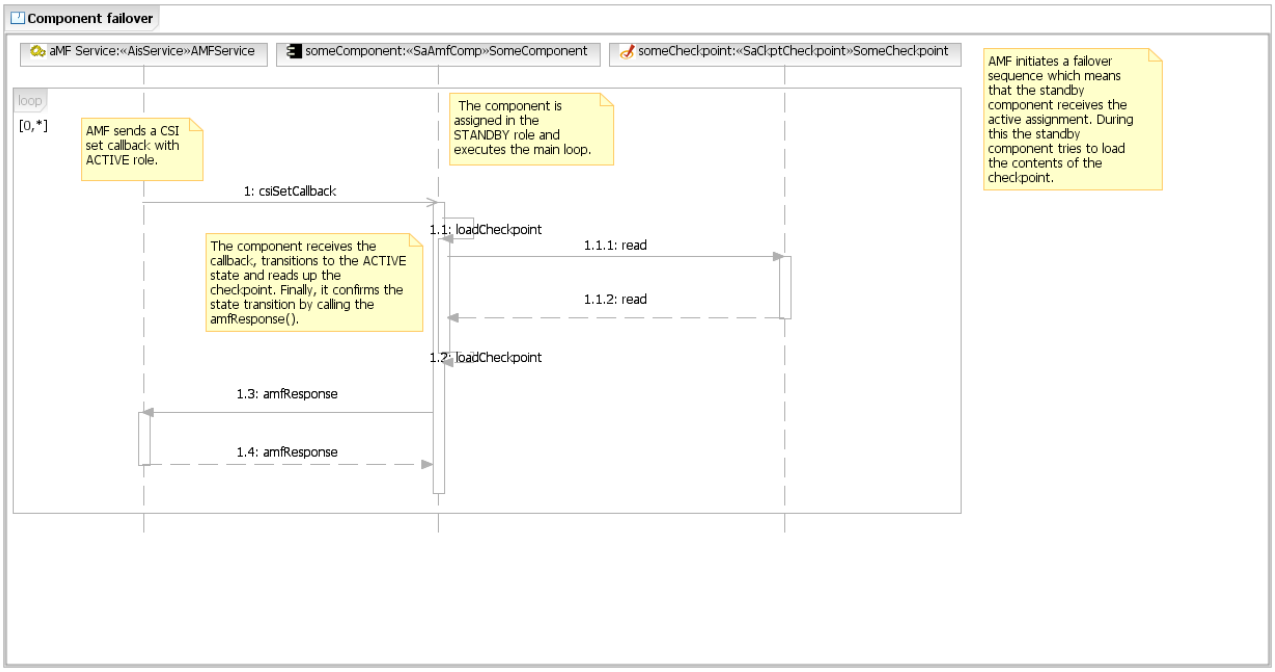


Figure 23 : Checkpointed Component Failover

Figure 24 shows the regular state save operation and its extension for the hot standby case. During state save, the component writes its internal state into the checkpoint. If the standby component is a hot standby one, which means it keeps its internal state continuously in synch with the active component to enable immediate service take over on failure, then after doing the state save, the standby component is notified through the *stateUpdateNotification()* that new state information is available. The standby component then reads up the new state information.

Figure 24 : Hot Standby Component State Save Scenario

2.4 Application Design Support

The modelling techniques described in Section 2.2 outlay the basics of application development. However, further support is required for the application designers and developers to ease and quasi standardize their work. This section presents an overview on the three application design support tools developed within the framework of the HIDENETS project i.e., the domain specific editor, the configuration generator and a code synthesis tool.

2.4.1 Domain Specific Editor

In order to support seamless modelling and development activities for the HIDENETS platform we constructed a domain-specific editor built on the advanced profile handling capabilities of the IBM Rational Software Architect (RSA, see [RSA]) environment. RSA enables us to assign visual notation to stereotypes (icons or even entirely new shapes) and provides a straightforward lightweight extensibility mechanism through "pluglets" (typically used for performing some basic well-formedness checking on software model (e.g., in our case we can check that a class marked as a client of a cooperative backup service is actually connected to a cooperative backup storage).

Below we present some screenshots taken from the domain-specific editor. The examples were intentionally designed to show the same model fragments as the application examples discussed above. The stereotypes used in the domain specific editor are the ones discussed above prefixed with "hi" (HIDENETS) to prevent namespace clashes. The icons and images used by our domain specific editor were newly drawn by us, obtained from free sources or simply re-used Eclipse icons.

2.4.1.1 User Interface

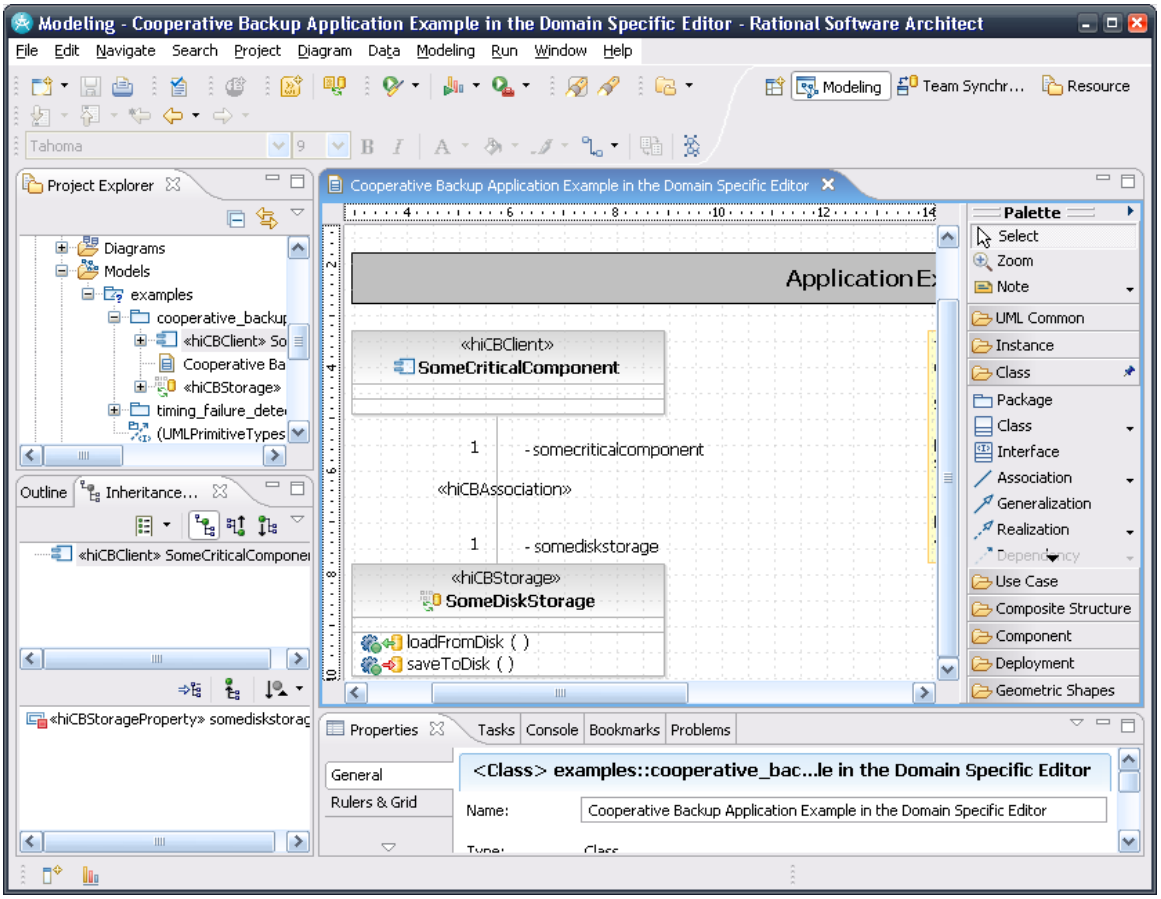


Figure 25 : Overview on the User Interface of the Domain Specific Editor

Figure 25 shows the user interface of RSA working as our domain specific editor. It is easy to see that applying the HIDENETS profile to a model and enabling the visualization features of domain specific editing does not disturb the well-known interface.

2.4.1.2 Application Examples

Figure 26 presents the application example for cooperative backup activities (see Figure 5 for the original example using plain stereotypes). There are three novel visual notations in the figure:

- Cooperative backup storage is highlighted by an icon representing a hard disk and arrows representing the data interchange operation (we re-used version control icons here). Since there can be a large number of classes acting as clients of a cooperative backup facility we decided not to override the usual icons here for clarity.
- Store and retrieve operations are depicted by small arrows targeting or originating in the hard disk symbol.

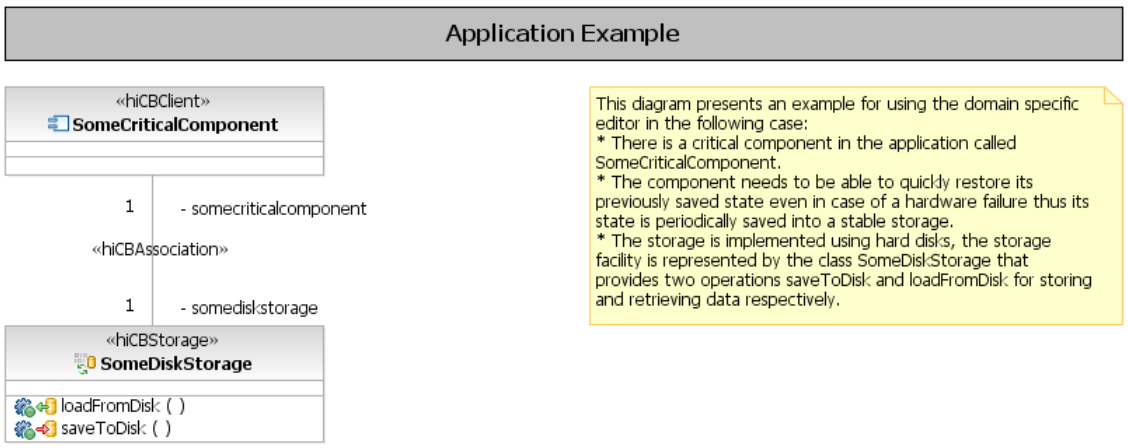


Figure 26 : The Cooperative Backup Example in the Domain Specific Editor

Figure 27 presents the application example for timing failure detection (see Figure 11 for the original example using plain stereotypes). Some of the novel visual notations shown in the figure:

- Real-time service provider class *MyRealTimeServiceProvider* is highlighted by a dark clock icon. (The real-time service client class has no special stereotype due to the considerations mentioned above.)
- The actual real-time service (method *callMe*) is indicated by a light clock icon.
- The real-time service agreement between the provider and the client is indicated by a clock and a checklist icon entirely replacing the association class shape.

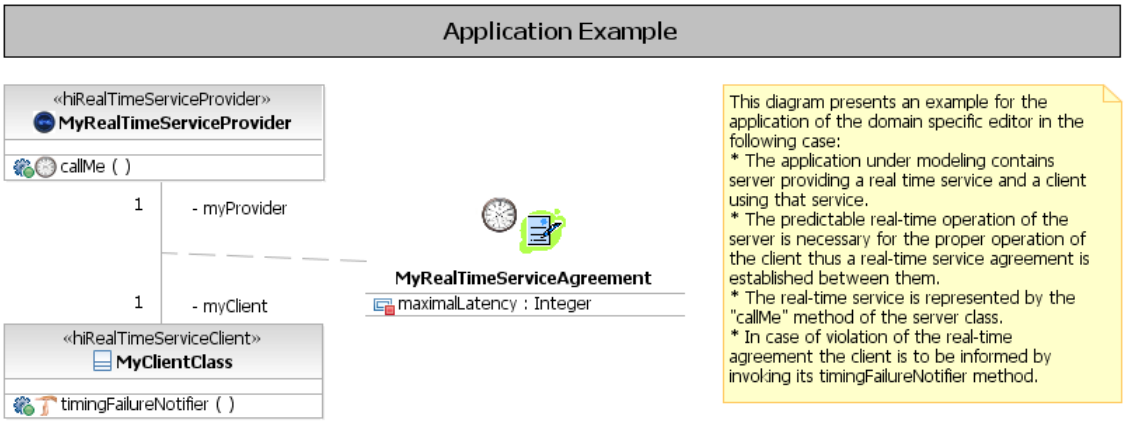


Figure 27 : The Timing Failure Detection Example in the Domain Specific Editor

2.4.1.3 Visual Notations in Explorer Views

Having assigned icons to key profile entities the same icons are used in various model explorer views, e.g., in the tree view of the RSA Project Explorer shown in Figure 28.

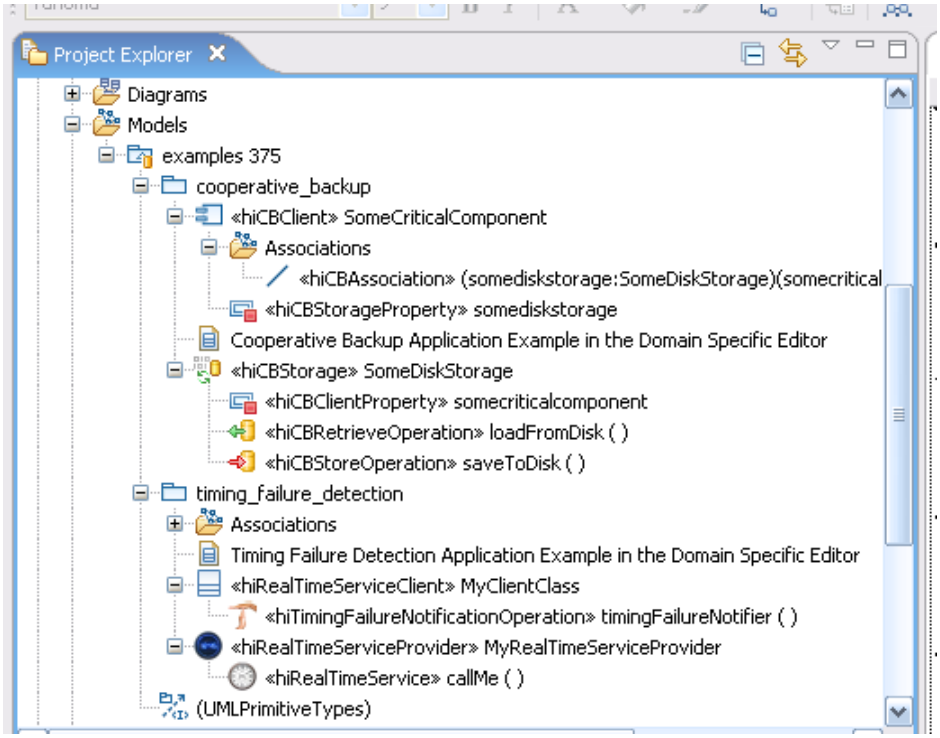


Figure 28 : Indication of Stereotyped Classes and Attributes in the Domain Specific Editor

2.4.2 Source Code and Configuration Generation

The AIS UML profile which is part of the HIDENETS UML profile can be used by any UML modeller that supports the usage of profiles. However, only attaching the stereotypes to the elements will not make it easier to create understandable application/system models. The following types of diagrams may be used for thematic visualization of the application and system models.

- The **Resource view** visualizes the relations of AMF resource type entities, such as the Application, Service Group, Service Unit and Component.

- The **Architecture and Deployment view** shows the Cluster and Node objects, and the assignment between the resources and these nodes. Basically, it describes the cluster and the deployment of the different resources (Service Units, Checkpoints…).
- The **Service view** is used to represent the service entities of the model, like Service Instance and Component Service Instance.

2.4.2.1 Example Application

In order to demonstrate this modelling approach, an example platoon monitoring system development is described in the following using the proposed framework. Monitoring is a very important key factor in many fields. This is no different in the vehicle fleet management. The events that the system has to observe are very rare and usually happen in a short period of time. So one of the main requirements for a monitoring system is that it provides its services in a highly available manner since the unavailability of these services can cause extremely costly or even unrecoverable results.

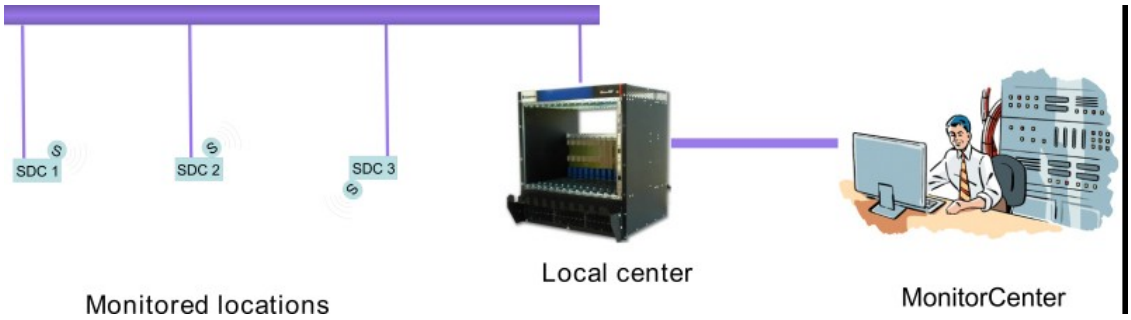


Figure 29 : Monitoring System

Figure 29 shows the structure of the system that consists of sensors to monitor different attributes of the platoon (velocity, position) and provides an administrative console to operators to visually monitor and control the current status of the system. The application services that process the measurement data and control the actuators – such as the fleet management system and platoon command centre – do run in the Local Service Center.

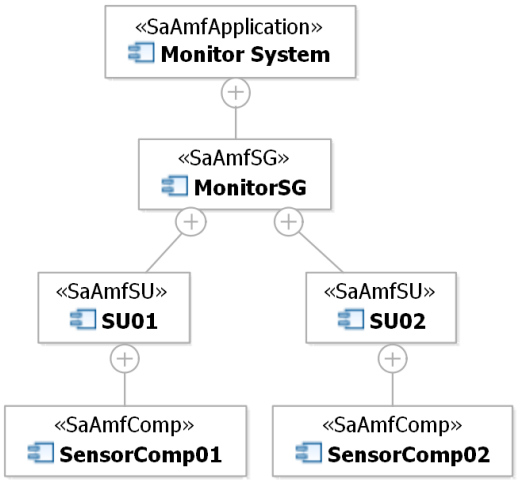


Figure 30 : The Resource View of the Model

The Figure 30 shows the resource view of the Monitoring System service model. The Monitoring System is represented by a component stereotyped *SaAmfApplication*. The application comprises

one service group that consists of two redundant service units which are handled according to the active-standby (failover-pair) redundancy scheme. Figure 31 shows that the service is realized using two nodes that compose a cluster. The Monitoring System application is assigned to the cluster and the *Service Units*, that contain the service provider components, are deployed on the nodes.

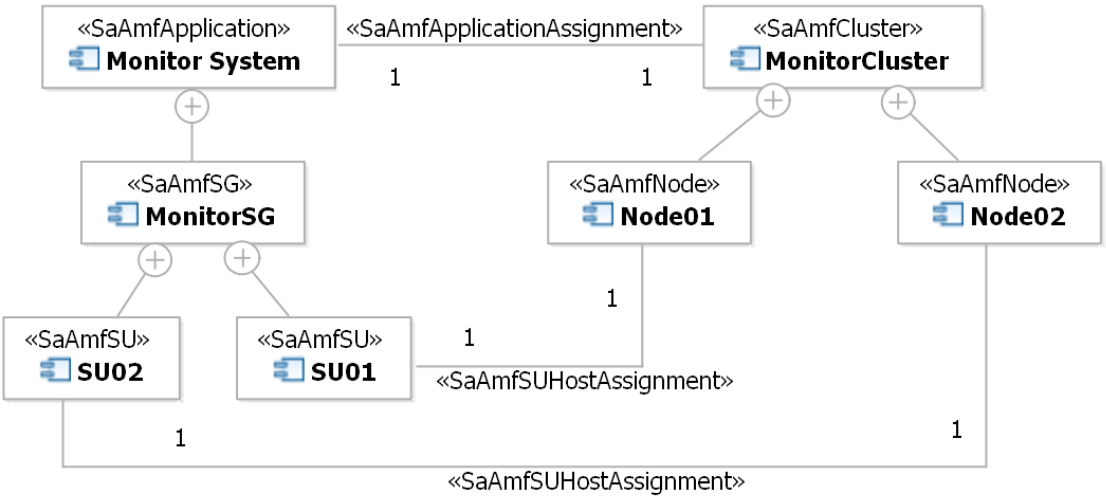


Figure 31 : The Deployment View of the Model

The service view of the model is depicted in Figure 32. The figure shows the service responsible for monitoring the sensors by the *<<SaAmfSI>>SensorSI* component.

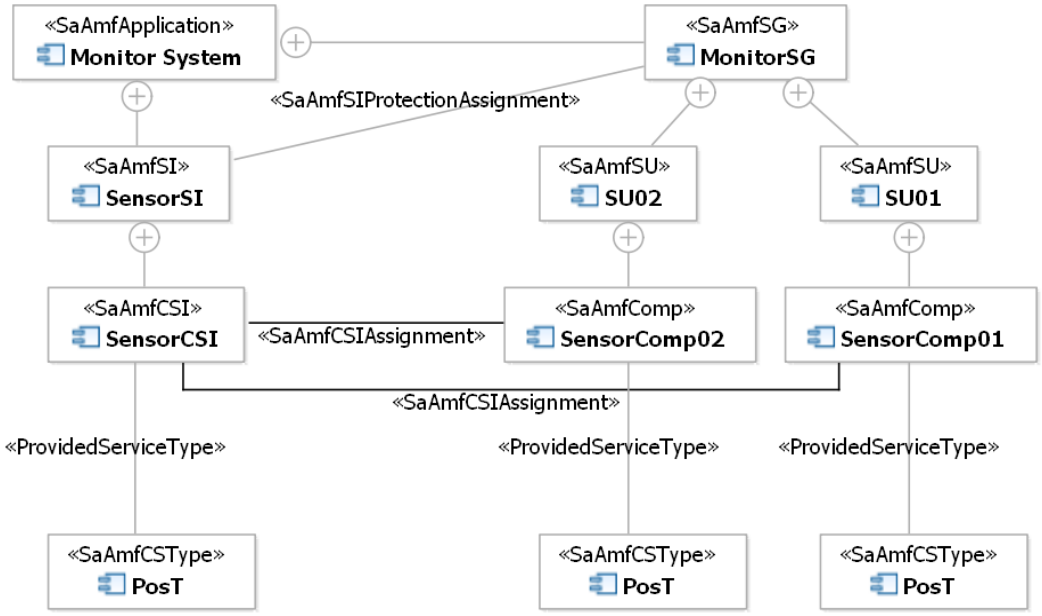


Figure 32 : The Service View of the Model

It is important to make sure that all services are supported by the Service Group they are assigned to, thus, the assignments of the CSIs comprised in the Service Instance are calculated automatically during validation. The figure shows the result of this step when associations between the *<<SaAmfCSI>> SensorCSI* and the available components are calculated.

This example application model will be used throughout this section for introducing configuration and code generation facilities of the HIDENETS framework.

2.4.2.2 Implementation of the Framework

The implementation of the framework is based on the *IBM Rational Software Architect* (RSA) modelling product, which is built on the extensible architecture provided by the Eclipse open source development platform.

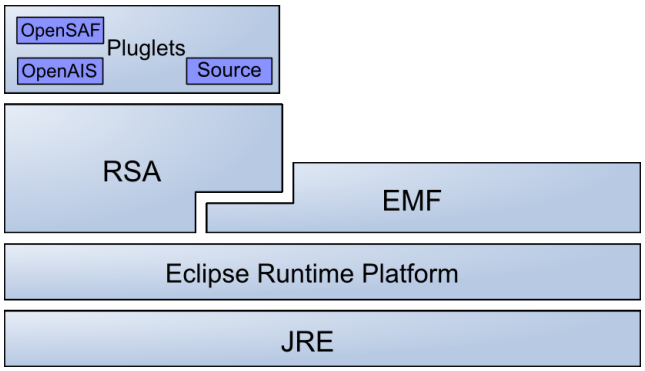


Figure 33 : The Architecture of the Rational Software Architect Platform

The RSA provides a modelling framework for standard UML models and supports the creation and application of UML Profiles on those models. Additionally, it provides means for accessing and modifying the models in a programmatic way through specific APIs.

The Eclipse environment that the RSA operates in enables a very high level of extensibility through plugins. Actually, the RSA itself is a great number of plugins in the basic Eclipse environment too. Similarly to plugins, the RSA supports an even more lightweight extension mechanism through *pluglets*. Pluglets are Java applications integrated into the RSA framework, and they are able to access the application model in the model space using the *Eclipse Modeling Framework* (EMF) APIs.

2.4.2.3 Model Manipulation

The EMF API provides basic level of access to the model in the model space. Model entities can be created, accessed, modified, deleted, etc. In order to support higher level of access to the service model, utility functions have been created. The most important ones are:

- **getApplicationObject(Object o)** iterates recursively the model space and returns the list of the objects that are stereotyped using the *SaAmfApplication* stereotype.
- **getClusterObject(Object o)** iterates recursively the model space and returns the list of the objects that are stereotyped using the *SaAmfCluster* stereotype.
- **getChildren(Component o, Stereotype s)** returns the list of components that are packaged in the given component *o* and are stereotyped using the *s* stereotype.

In Figure 34 the source code excerpt demonstrates the usage of the utility functions. First the cluster object is retrieved, then an application associated to that is selected, and finally the application configuration is recursively generated by enumerating the contained elements.

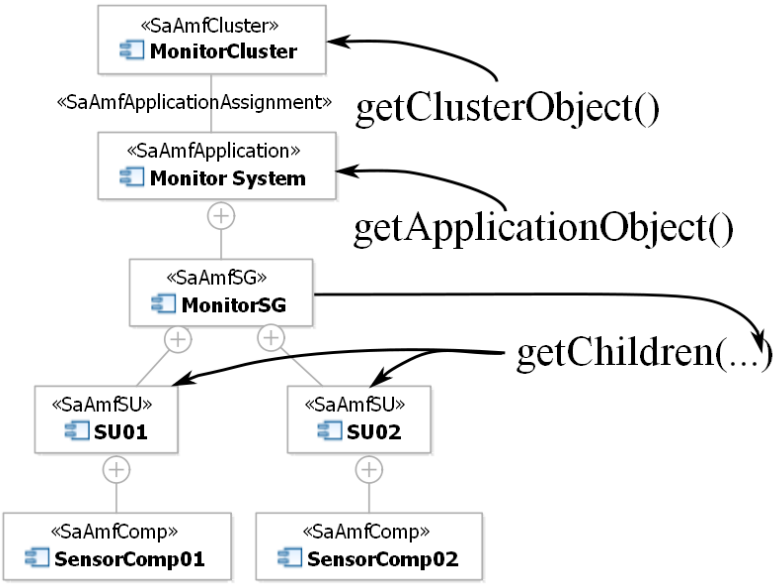


Figure 34 : Sample Code for Model Access Using High Level APIs

2.4.2.4 Configuration Generation

One of the main objectives of our Framework is to help the creation of AIS implementation specific configuration descriptors. For this purpose, we use the application models defined in the modelling tool.

The configuration generator facility has to be implemented for each different AIS implementation in the form of RSA pluglets. In the following, we introduce the pluglets created specifically for the OpenAIS and OpenSAF middlewares.

Configuration for the OpenAIS middleware

OpenAIS [OpenAIS] is a simple and easy to use SA Forum compliant middleware. It stores the configuration data using a simple text file. Using the high level utility functions described above, the application model is traversed and the corresponding configuration data is generated.

Although the model describes the AIS service, not all of the attributes in the configuration file can be generated automatically. There are a few specific attributes, e.g. the path of the binary executables that should be manually set after the generation of the configuration file.

The OpenAIS configuration of the example service described earlier is generated using the developed pluglet. Figure 35 shows the application model and the generated configuration. Whenever the pluglet finds an attribute not specified in the application model, it uses the *TODO* tag to indicate that the particular value should be given manually.

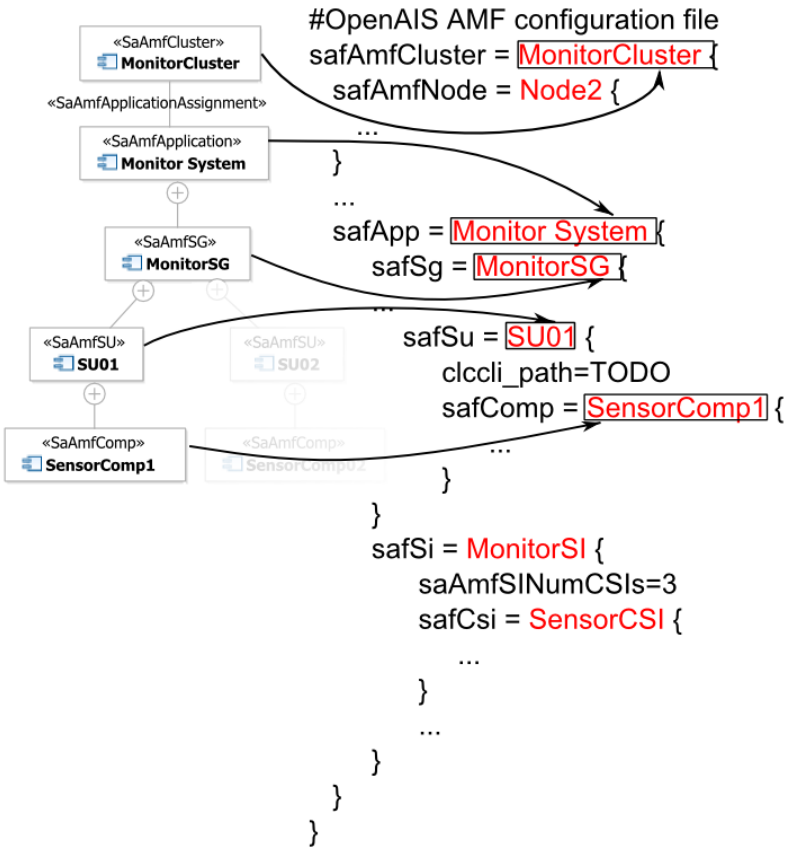


Figure 35 : The Generated OpenAIS Configuration File

Configuration for the OpenSAF middleware

While OpenAIS defines a simple text configuration file, OpenSAF [OpenSAF] uses an XML structured configuration file. In Java there exist a number of solutions for generating XML files:

- *plain text* method is a low level solution,
- the *DOM* provides standard XML construction, classes
- and a high level solution is provided by the *JAXB*.

For our implementation, we selected the *DOM* solution, since it is supported in all Java environments.

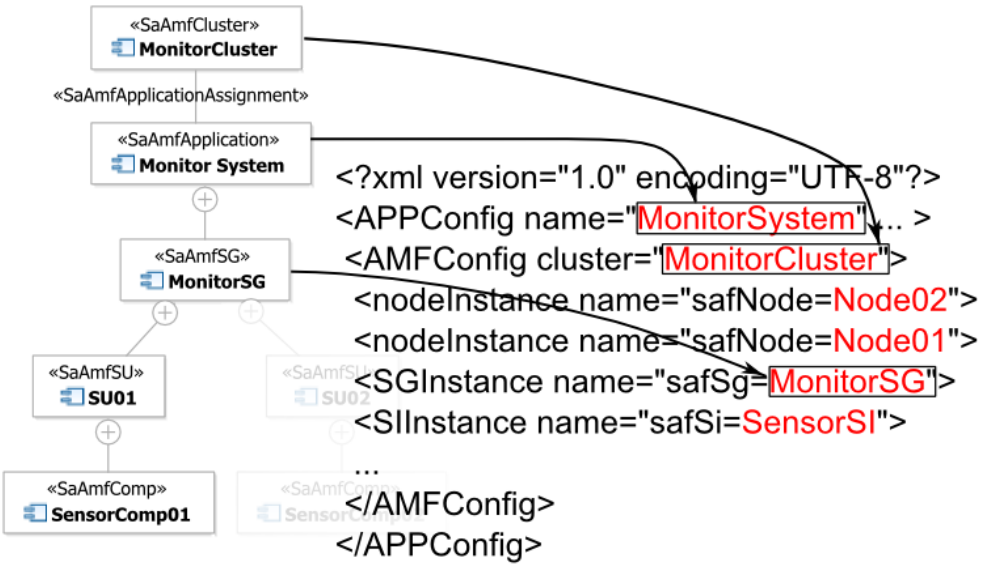


Figure 36 : The Generated OpenAIS Configuration File

Figure 36 shows a part of the generated configuration data for the OpenSAF middleware of the example application.

Component code skeleton generation

Similarly to the configuration data, the skeleton of the component source code can be automatically generated based on the constructed application model. Each component has similar structure and thus all the generated entities can be created from a common code template.

As the first step, we analyzed the structure of AMF components in order to create the general component code, and then based on our experience with configuration generation, we developed a source code generator pluglet.

Structure of the code: Every AIS service has its own purpose but all of them are specified according to the same programming conventions. This common functionality can be characterized as lifecycle handling (initialization, finalization) and event dispatching for asynchronous operation. Based on these patterns, the following sections can be found in the general components source code:

- Lifecycle methods for every AIS service
- Event loop for dispatching events
- Callback method stubs that are called on events
- Component specific configuration attributes (e.g. AIS version number, healthcheck keys)
- Business logic methods

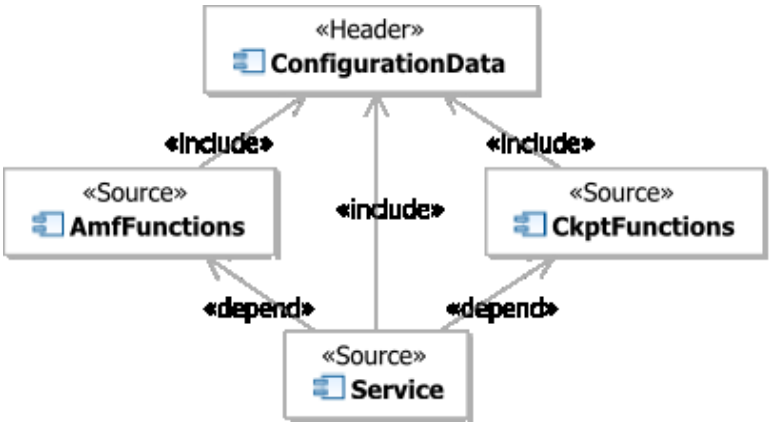


Figure 37 : The Structure of the Source Code

The structure of the source code is depicted in Figure 37. Modules are created for every AIS service in the form of C source files, and a common header file, which defines the component specific attribute values. These modules have certain dependencies, like all source files include the header file and functionality of the corresponding AIS service modules.

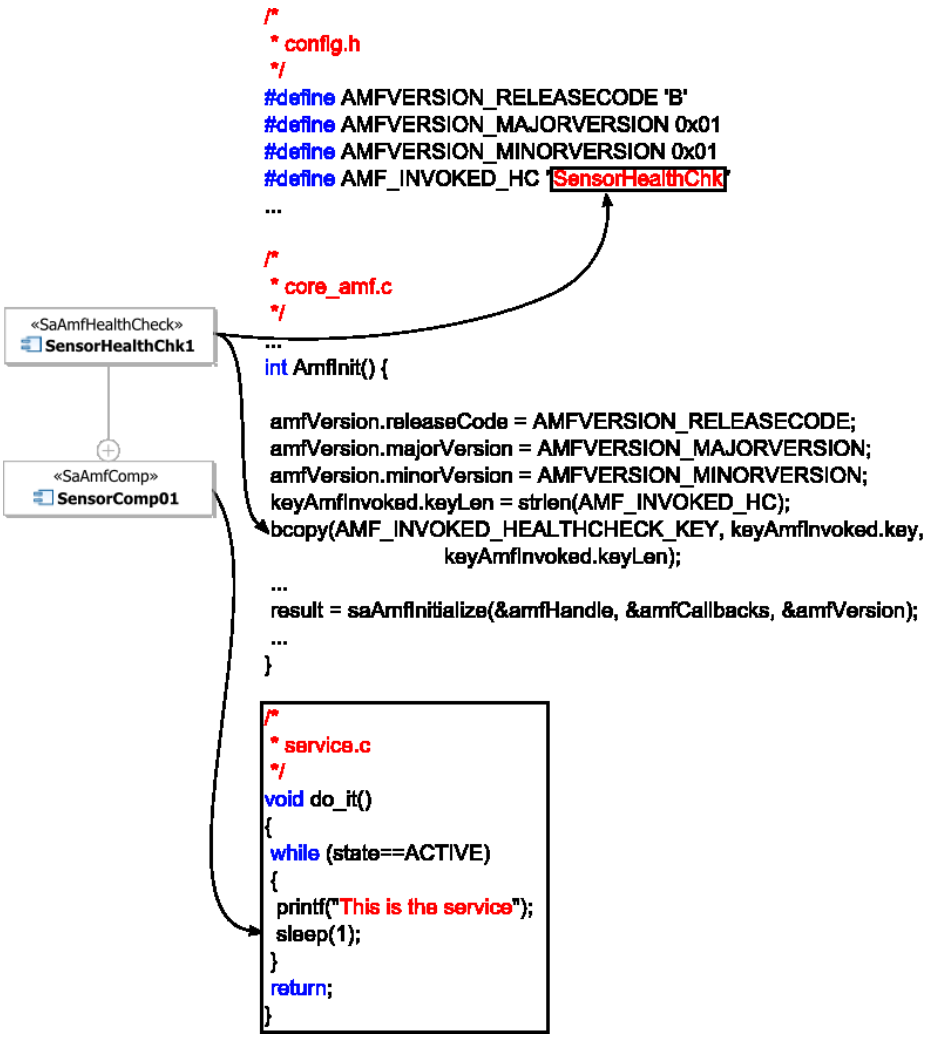


Figure 38 : The Generated Source Code

The code generation: The code generator pluglet was created similarly to the configuration generators. It uses the general component template defined above.

Figure 38 shows the modules generated for a component. The generated source code contains three major parts:

- the configuration header,
- the AIS service modules and
- the service itself.

This particular component (*SensorComp01*) has one healthcheck (*SensorHealthCHK1*) associated with it.

Every component specific attribute manifests as a compiler definition in the configuration header. For example, the key of the healthcheck appears in the configuration header as *#define AMF_INVOKED_HC "SensorHealthChk"*.

The executable component is built up by compiling and linking all these modules and the AIS libraries together.

2.5 Proof of Concept

This chapter introduces the reader to the practical benefits of the aforementioned efforts, providing an example application over the HIDENETS platform that was developed as a WP6 activity, namely the Application Development Test-bed (ADTB), extending its previous deliverables with an insight to the model-driven nature of those activities.

The Proof of Concept chapter is organized as follows. Section 2.5.1 introduces the application giving an initial notion on its specifics, the entities taking part. Section 2.5.2 gives a good insight into the use-case driven nature of MDD resulting in a functionally decomposed software architecture. That is followed by the utilization of the underlying metamodel(s) and profile(s) of HIDENETS in section 2.5.3, while 0 shows the benefits of the previous steps during the implementation of the application.

For further details on ADTB specification, implementation and evaluation the reader is kindly directed towards [D6.2], [D6.3] and [D6.4] respectively.

2.5.1 Application Development on a Conceptual Level

The initial goal of ADTB was (1) to show the benefits of the basic resilience mechanisms of HIDENETS and (2) to give an example of using COTS solutions in utilizing SA Forum specifications and middleware. Therefore we choose one of our safety-related use-cases from [D1.1] where both car-to-car and car-to-infrastructure communication is involved

We decided to implement a Platoon Driver Support System (PDSS) application that is a simplified version of the Platooning scenario [D1.1]. The implementation of a safety critical, hard real time application like Platooning involves quite some extra efforts in an embedded environment using direct links between components to ensure timeliness which does not completely match our scope and intention. Therefore we decided to simplify it by

- removing hard real time requirements and
- reducing the application to a driver support facility instead of suggesting that our application would be capable of actually replacing the driver.

Obviously the application we would like to develop is primarily a demonstration: the key contribution of our work is not the application as software product, but the *development process itself*.

2.5.1.1 Concepts

The context of the PDSS application is less critical than in case of a full featured Platooning software (in the context of HIDENETS use case scenarios in [D1.1] this application can be defined as a combination of parts of the Platooning and the Floating Car Data scenarios): in our case all vehicles in the platoon are driven by human drivers, the PDSS application serves more like an intelligent cruise control and platoon management software. The idea is as follows: the first vehicle in the platoon is the platoon leader (called *Head Vehicle* or *HV* in this discussion) and the vehicles following it (called *Slave Vehicles* or *SV* in this discussion) should adjust their speed according to the head vehicle. This is supported by the PDSS software by collecting various parameters of vehicles (speed, acceleration, etc.) and calculating the necessary actuations to be applied (acceleration or braking) according to the reference head vehicle. This function of the application only provides an intelligent support for human drivers of slave vehicles, the drivers may choose not to use this service. Another feature of the software is to provide up-to-date information about the traffic conditions in the area to the driver of the head vehicle and periodically send the actual position of the platoon to the traffic administration centre (Infrastructure).

2.5.2 Functional Decomposition – Actors & Use-Cases

Now that we have the main concepts of our PDSS application, we have to *identify the actors and use-cases*[2]. In this step we identify (i) *direct actors* (i.e., human users or external systems that submit requests to the system) and (ii) *direct use cases* (i.e., the high abstraction level description of requests submitted by direct actors) and collect those (iii) *lower abstraction level use cases* and (iv) *external systems* that are used by direct use cases (no in-depth discussion just collection). The short summary of the specification below highlights these concepts in the context of the PDSS application:

The platoon consists of two or more vehicles. All vehicles are driven by human drivers whose work is supported by the PDSS application. The PDSS application presents the *driver of the head vehicle* (direct actor) with some important information by *displaying traffic data* (direct use case) related to the actual area. The PDSS software periodically collects data from the behaviour of both vehicles and calculates the optimal acceleration/braking etc. values to be applied to slave vehicles to adjust their speed to the head vehicle; this feature can be *enabled or disabled* (direct use case) by *drivers of slave vehicles* (direct actor). At the infrastructure side the company maintains an up-to-date database of traffic conditions that may be important for the platoons (e.g., road reconstruction works, accidents, special traffic conditions, etc.) and the actual position of platoons. Information about traffic conditions are *updated in the database* (direct use case) by a *transportation manager* (direct actor) of the company. She/he can also *display the actual position of platoons* (direct use case) based on the database. The movement of platoons is automatically reported by the PDSS software and stored in the database.

Below we present the direct users of the system with their short definitions and the services accessed by them in a tabular form.

[2] The terms „use-case” and „actor” have formally defined meanings in the UML. In this document we refer with these words to the corresponding UML model element types. Generally, the same terms may be used in a less strict sense, as it is the case in other HIDENETS deliverables, for example in [D1.1], as well.

Actor	Short definition	Use cases accessed by the actor
Head Vehicle Driver	Driver of the head vehicle. Her/his decisions are supported by the PDSS application (e.g., by the traffic information presented on a display) but otherwise the driver has full and exclusive control over the head vehicle.	Displaying traffic data Enabling or disabling PDSS control
Slave Vehicle Driver	Driver of a slave vehicle. Her/his task is supported by the PDSS application: when operating the PDSS software works as intelligent cruise control device that adjusts the speed of the vehicle according to the behaviour of the head vehicle. The slave vehicle driver may switch off the PDSS control either explicitly or by touching a pedal in the vehicle (in order to enable direct and immediate human interception in dangerous situations).	Enabling or disabling PDSS actuation
Transportation Manager	An employee of the transportation company being responsible for maintaining the traffic database. The transportation manager can also display the actual position of platoons.	Displaying platoon movement data Updating the traffic database

Investigating these direct use-cases, we find that underlying services (like communication between the HV and Infrastructure, i.e. “Traffic data communication”) and external systems (displays and a database) are necessarily involved in executing these tasks (see Section 2.5.1.1).

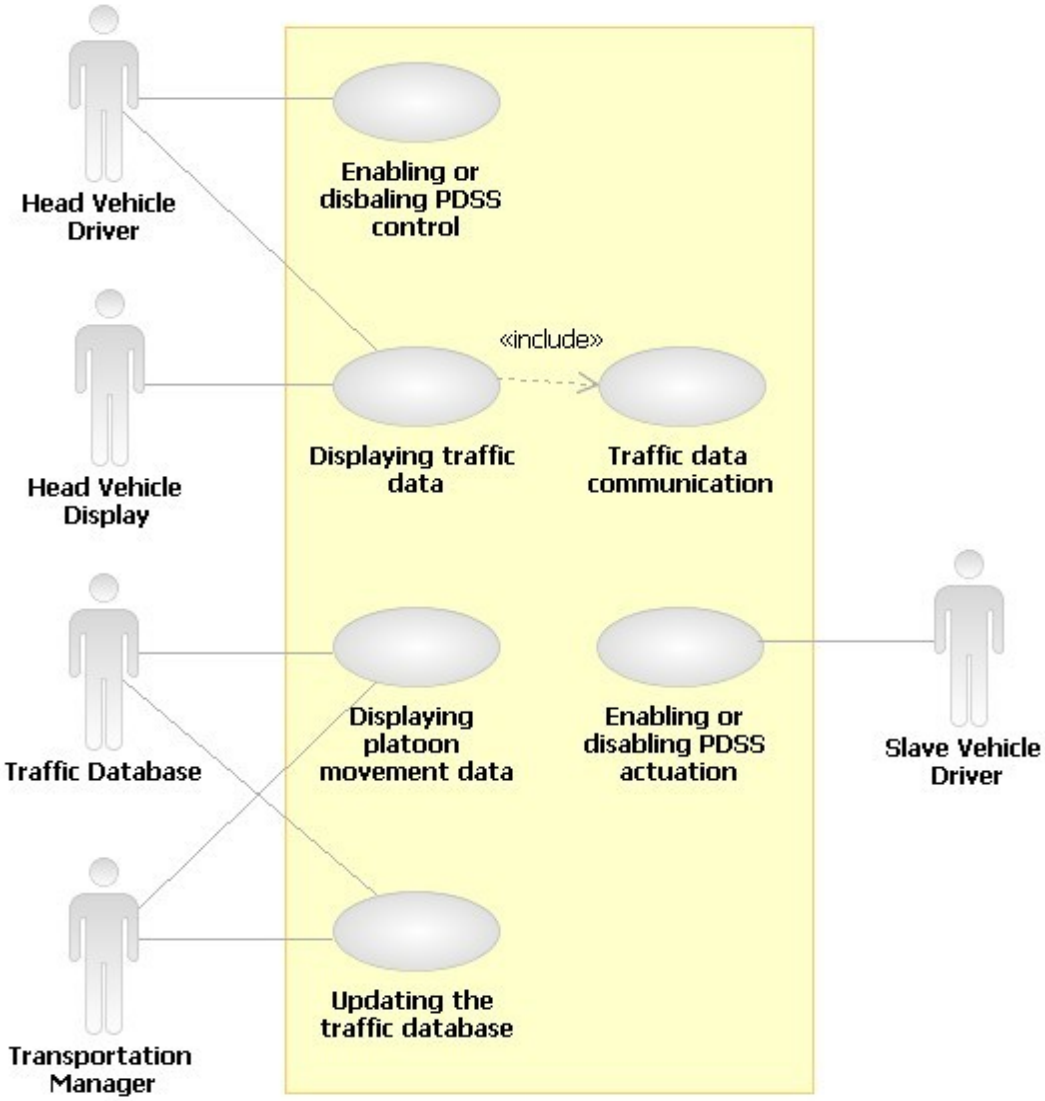


Figure 39 : Direct Actors and Use Cases with Required Lower Level Services

We have to continue in the very similar way: iteratively refining our specifications and thus defining *implicit actors, use-cases and lower level services*. The result is depicted in Figure 40.

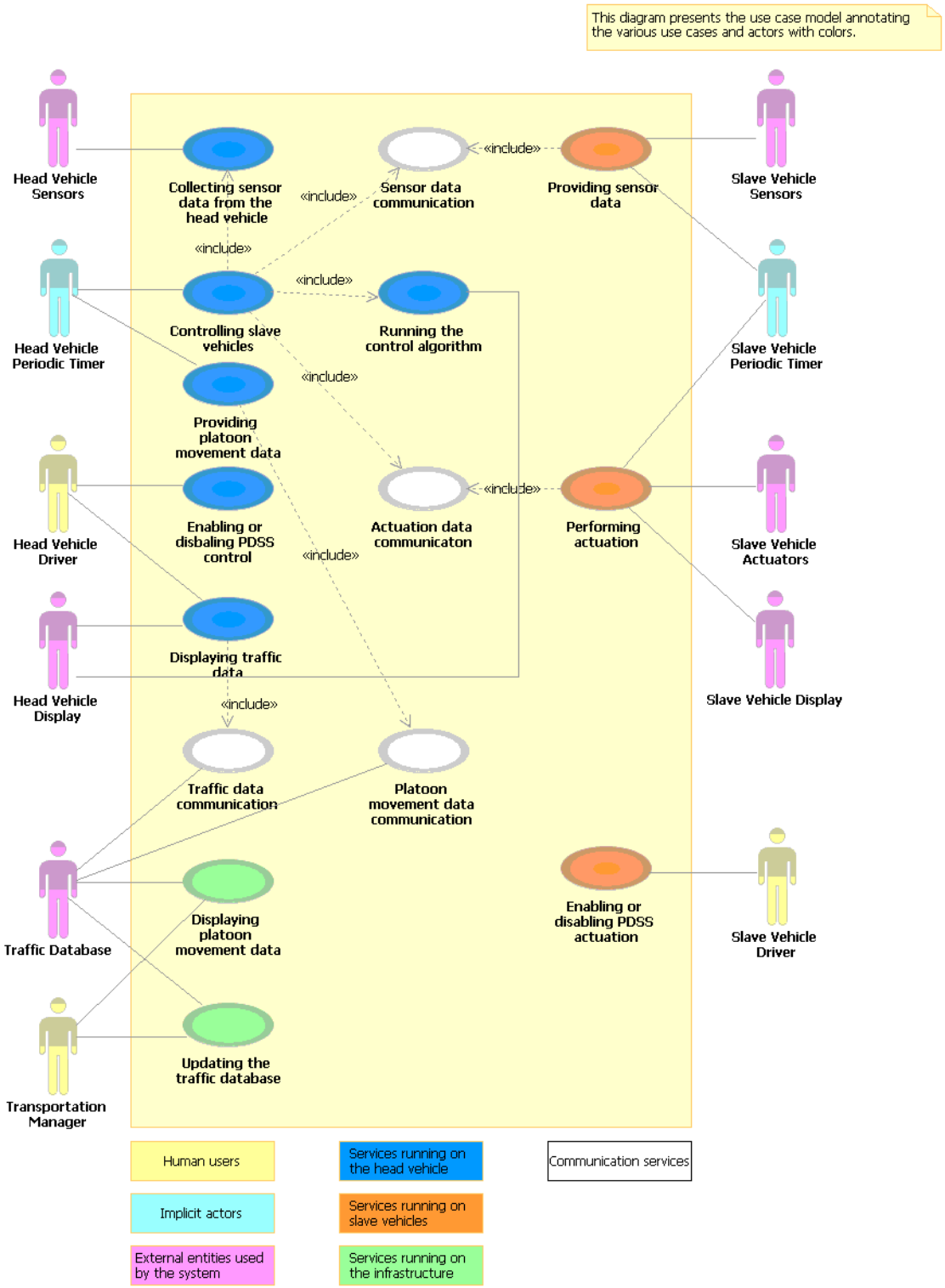


Figure 40 : A Color Annotated Overview on the Use Case Model

Having the use-case model, we can begin to design the application that will resemble the scheme above. The following Figures (Figure 41, Figure 42) demonstrate that trait of MDD.

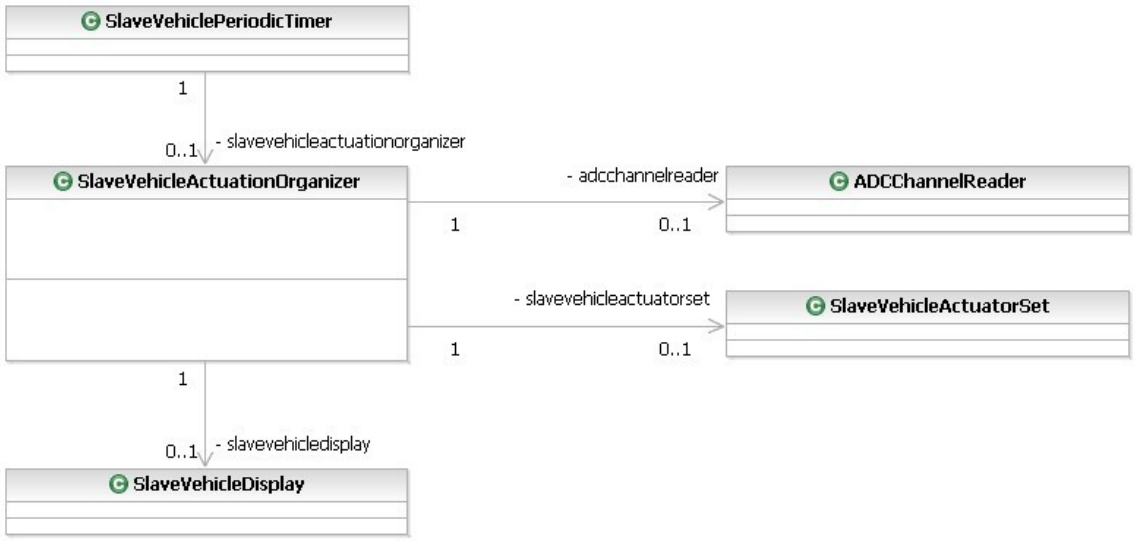


Figure 41 : Key Classes of Actuation at Slave Vehicles

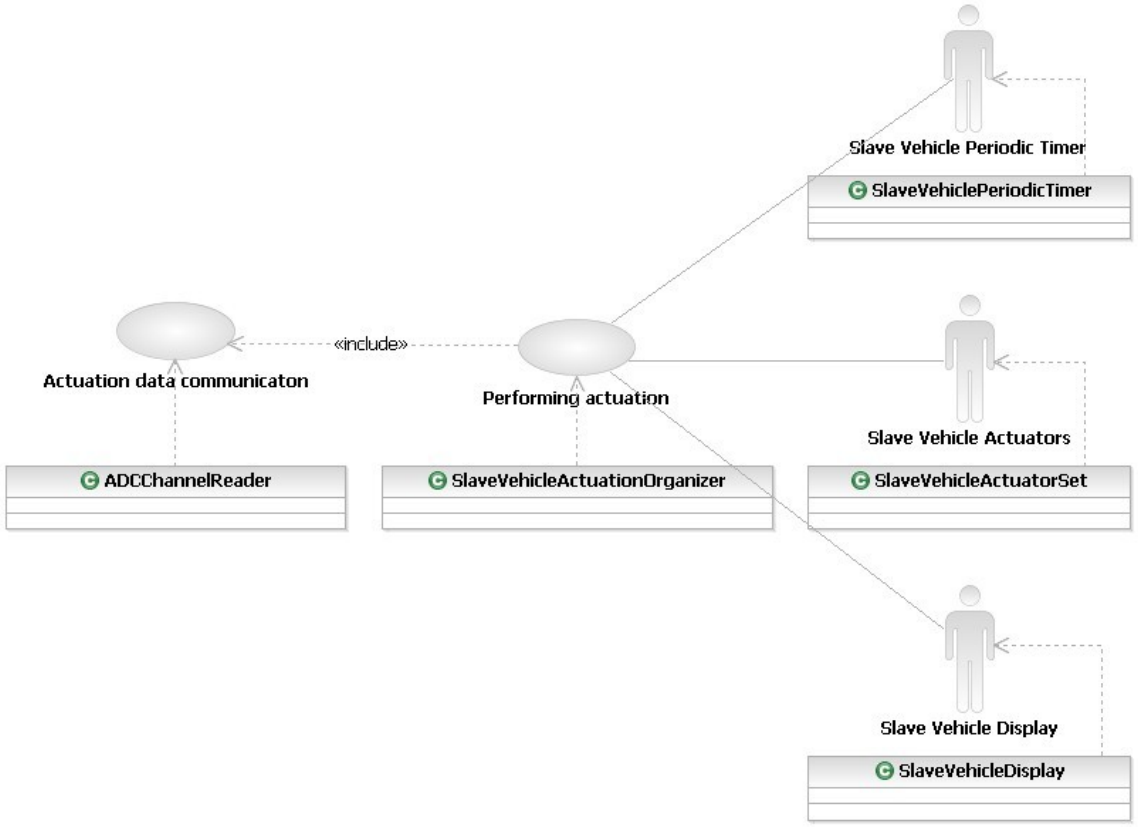


Figure 42 : Relation of Key Actuation Classes to the Use-Case Diagram

2.5.3 Utilizing the Underlying Metamodels

Now we can use our Domain Specific Editor (DSE) to design the application and utilize the built-in features. Main functionalities (use-cases) are aggregated into *components* that consist of the classes performing lower-level services. As for the previously used example, see Figure 43.

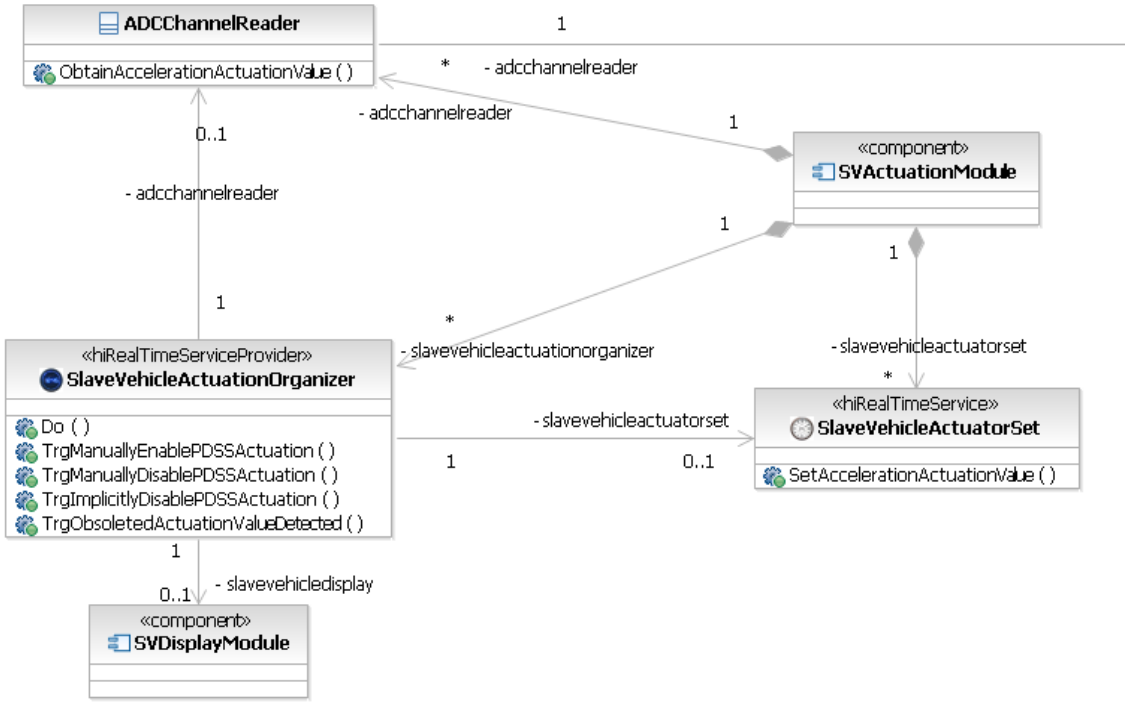


Figure 43 : Slave Vehicle – Actuation

SVActuationModule is tagged with <<component>>, aggregating the actual actuation functionalities as well as the necessary communication, while – through an association – it is also linked to the Display that also has to be periodically refreshed to keep the driver up-to-date. When speaking of periodicity, one has to consider the great importance that timeliness plays throughout the application. Timely events are triggered by timers (marked teal in Figure 43) in the PDSS application while their corresponding classes are stereotyped in DSE, making them not only easily identifiable in the application model, but also enforcing the usage of the appropriate design pattern and indicating their connection with the underlying TTFD service (see section 2.3.2.2 for further details).

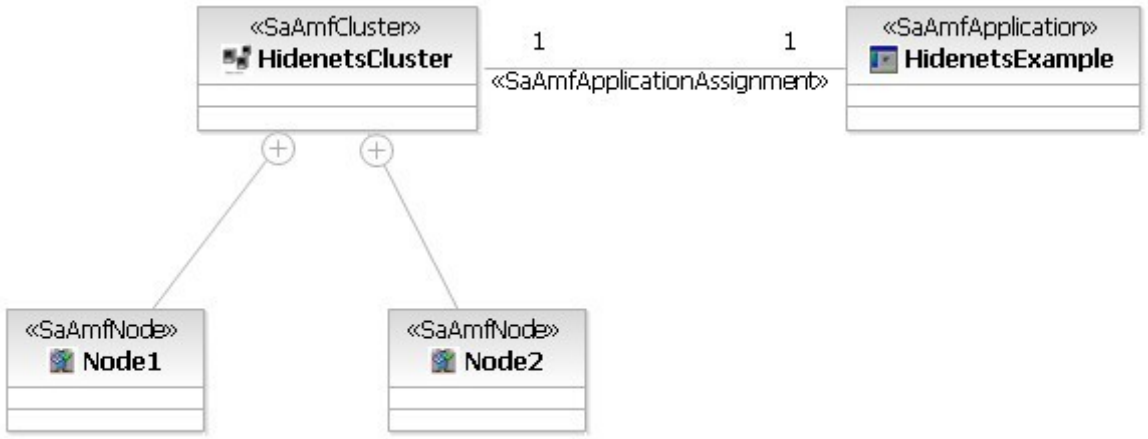


Figure 44 : Cluster Definition

As for the infrastructure domain, the task is more complicated. According to AIS specifications the definition of the cluster organization is highly important thus making it impossible to directly map

the use-case model into an application model, because use-cases here do not involve physical traits or deployment issues in particular, while at the same time they would be essential for configuration of applications running over SA Forum middlewares.

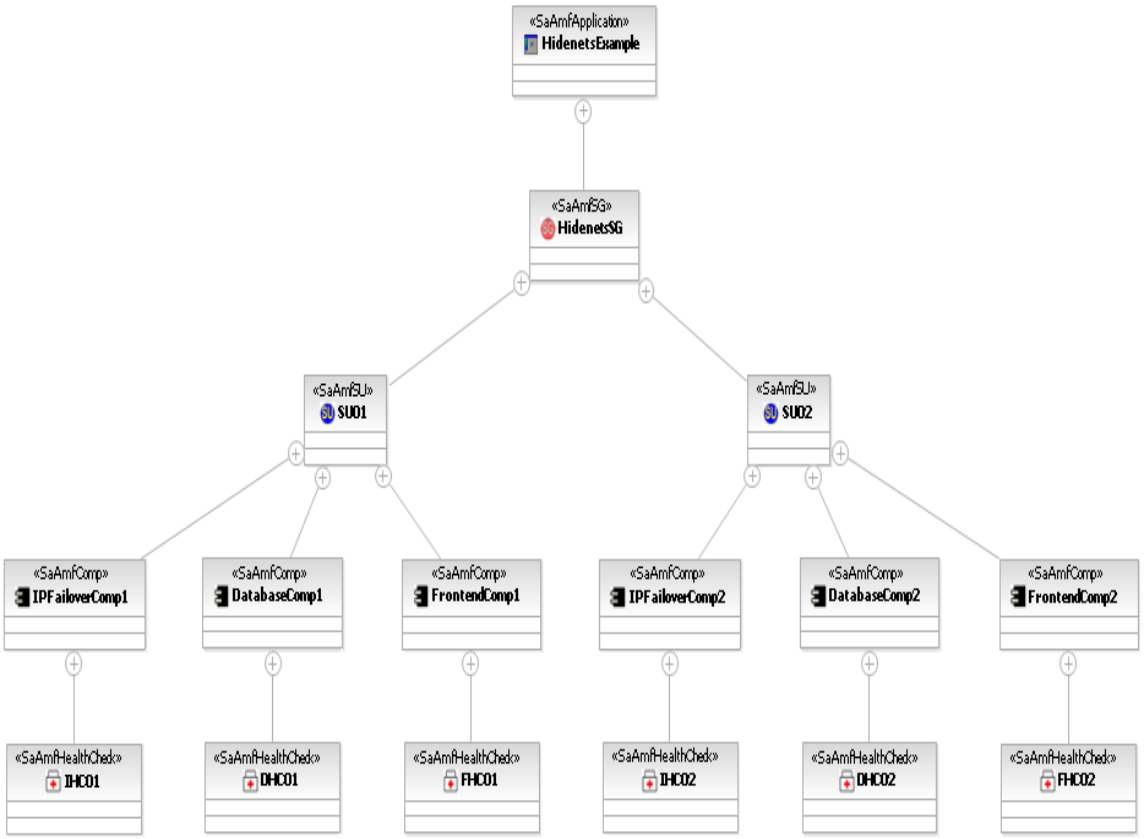


Figure 45 : AMF Entities

So we want our infrastructure domain components to run over a cluster consisting of two identical nodes. That way we defined the *physical deployment* of the application (Figure 44).

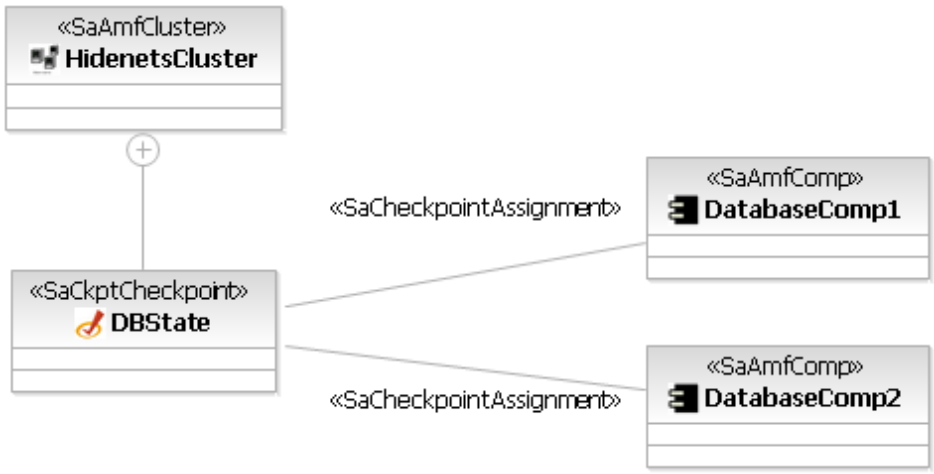


Figure 46 : Checkpoint Assignment

As a next step, we have to define the deployed application in the sense of the *Availability Management Framework* (Figure 45). Please note that the *logical representation* resembles that of

the physical, the application consisting of a single Service Group that is composed of two identical Service Units, each containing three AMF Components that together form our application in that domain and that are all continuously health checked. Whenever a *HealthCheck* returns failed, *IPFailover* is responsible for the cluster internal failover. The *Frontend* provides the visualization towards the traffic administrator while the operation of the underlying database is shared between the two components through checkpointing (see Figure 46).

2.5.4 Implementation

The ADTB is implemented in C language in both domains. In the ad-hoc domain we used the defined HIDENETS middleware, more precisely the *Reliable&SelfAwareClock*, the *AuthenticationService* and the *TimelyTimingFailureDetection*. These were implemented as stubs, mimicking the defined interfaces, but providing a simplified functionality. While invoking those functionalities we relied on the defined design patterns that way giving the code a uniform and easy to understand structure at those parts closely related to the middleware. As the experimental set-up is running on a laptop with virtual machines we used RPC calls in the communication of the "vehicles" to resemble communication delays as best as possible.

The infrastructure domain components of the PDSS application were built over SA Forum specifications using the open-source middleware implementation OpenAIS. The AMF components were implemented using the model-based code-generator while the deployment to the cluster[3] was done using the configuration-generator tool (see Section 2.4.2 and the diploma theses [Szat, Urb] for further details on code and configuration generation).

2.6 Conclusion

In the original project proposal, the work on "UML design patterns and workflow" has set an aim to

- elaborate an application development methodology that helps application designer in the understanding and effective utilization of the (dependability, mobility and communication related) domain knowledge that is manifested in the HIDENETS middleware,
- define a UML profile incorporating the peculiarities of this environment and allowing a semi-formal formulation of user requirements and basic architectural solutions, and
- formulating design patterns to support the direct reuse of the HIDENETS architecture and middleware solutions while application development.

This means, that our work focused on the development of applications of the HIDENETS middleware. After reviewing the state of the art in application development and studying existing standards in we have

- elaborated a model-based application development methodology, that is based on a multi-step approach (see Sect. 2.2.2),
- defined domain specific metamodels and profiles for the HIDENETS applications,
- specified an application development tool-chain (see Sect 2.2.3 and Sect 2.4),
- gave a set of design patterns to help the use of HIDENETS services for application developers, and

[3] The cluster was provided by Fujitsu-Siemens Computers in the form of a TX120 office server and three Lifebook E8410 laptops.

- supported the modelling work of other work packages of the project.

Beyond this with our work we have contributed to an existing standard. The SA Forum standard interfaces were chosen as a basis for the middleware architecture of HIDENETS nodes in the infrastructure domain, and our results in supporting application development (metamodels, application development methodology, code and configuration generation, mobility related results) were presented to the SA Forum community, where they aroused general interest, and were received as valuable contribution and basis for ongoing standardization activities in the corresponding fields. Based on these results, FSC, BME and other partners in the SA Forum plan to continue the enhancement and extension of the SA Forum standard interfaces beyond the time frame of the HIDENETS project.

Furthermore, our work has demonstrated the wide extensibility of UML and the benefits of applying model based techniques. Building formal models of the HIDENETS node architecture was a necessary step in our task, while these models became a very useful tool in the project-intern communication (documentation of the services, discovery of dependencies of services, consistency and completeness checking, ...) for an international project team.

3 Testing Activities

Software testing consists of executing a program with some valued inputs and then verifying whether the outputs conform to the expected behaviour. In this section, we address the challenges and methodologies for the verification of HIDENETS-like applications and middleware services using testing. This contributes to the following project goal:

"Identify development tools and mechanisms like design patterns and testing methodologies to assist in the implementation of said service qualities." (from "HIDENETS – Description of Work") [HDoW]

A summary of the contribution is first given in Section 3.1. Then, a more technical presentation is developed in Sections 3.2 to 3.3.

3.1 Summary of the testing contribution

Work was focused on the verification of the highest layers in the HIDENETS architecture, that is, the application layer and possibly some high-level middleware services. We consider functional (black-box) approaches to test whether applications fulfil their expected requirements. Note that quantitative evaluation, e.g., reliability or availability assessment, is not addressed here (it is studied in Deliverables D.4.1.2 and D.4.2.2 [D4.1.2, D4.2.2]). Our interest is on the correctness issue.

As a first step, a review of relevant literature has been performed together with a testing case study that allowed us to gain concrete insights into validation problems. The results were detailed in Deliverable D5.2 [D5.2] and published in [Mics, Waes]. One of the conclusions was the lack of adequate formalisms to capture system-level behaviour and spatial topology in a mobile setting. Work has then been directed toward the definition of a scenario-based testing framework [Ngu] that covers (1) the definition of a language that describes interaction scenarios in mobile settings, and (2) some automated support to analyze and implement scenarios on a test platform with simulation facilities.

This section introduces the scenario-based testing framework developed in HIDENETS, and gives a high-level view of the underlying technologies.

3.1.1 Role of scenarios in the testing framework

Scenario descriptions are useful to support various test-related activities, such as the representation of requirements, of test purposes (i.e., interaction patterns to be covered by testing), of test cases, or of execution traces. Accordingly, the testing framework depicted on Figure 47 shows scenario-based artefacts that may be produced during different V&V phases.

This testing framework does not require commitment to heavyweight formal methods. Hence, the transition from one test specification artefact to the other may be informal, as expressed by the dotted lines. For example, test purposes may be derived informally from the important requirements, and test cases may be proposed by the user to cover some intended purpose. Note that the framework does not *preclude* the use of more formal approaches. Would a complete specification of behaviour be available, then the framework could possibly be extended to support formal treatments such as: the verification that the behaviour model exhibits the requirement scenarios, or the automated generation of test cases from a model and a set of test purposes. However, such formal treatments were not investigated within HIDENETS.

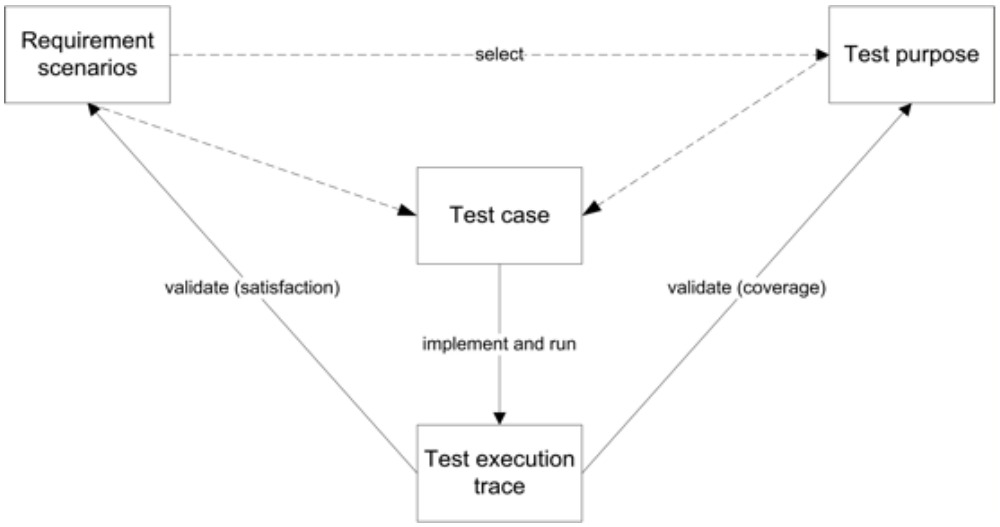


Figure 47: Overview of the testing framework

Even if a complete specification of behaviour is not available, some automated treatments become possible by simply using scenario descriptions. They are indicated by solid lines in the figure and are the focus of our work. The treatments include:

- Checking whether a test execution trace satisfies a requirement scenario.
- Checking whether a test execution trace covers a test purpose.
- Assisting in the implementation of test cases (and more specifically in the production of concrete contextual data).

Key issues are then the choice of an adequate scenario language, and the definition of a clean semantics allowing the above treatments.

3.1.2 Specificities of scenarios in mobile settings

A number of scenario languages have been proposed to represent interactions in distributed systems, like Message Sequences Charts [MSC] or UML Sequence Diagram [UMLsup]. They are, however, not sufficient to account for mobile settings.

Mobile computing systems, such as the ones targeted by HIDENETS, involve devices (handset, PDA, laptop, intelligent car…) that move within some physical areas, while being connected to networks by means of wireless links (Blue-tooth, IEEE 802.11, GPRS…). Such systems, and more specifically systems with applications in the ad hoc domain, can be distinguished from "traditional" distributed systems by the following aspects:

- *Dynamicity of the system structure*. The number of mobile nodes is not fixed. It varies over time, due to the dynamic creation, suspension or shutdown of nodes. Besides that, connectivity between nodes is also highly dynamic. As the nodes are free to move arbitrarily, they can join or leave the system in an unpredicted manner. Links may be established or destroyed, yielding an unstable connection topology. The topological changes may be constrained by a mobility model (e.g., vehicles move in one-way or two-way direction, speed is bounded…).
- *Communication with unknown partners in local vicinity*. In mobile ad hoc networks, a natural communication is local broadcast. It is used as a basic step for the discovery layer in mobile-based applications (for example, group discovery service for membership protocols, a route discovery in routing protocols…). In this class of communication, a node broadcasts a message

to its neighbours. As the topology of the system is unknown, the sending node does not *a priori* know the number and identity of potential receivers. Whoever is at transmission range of the sending node may listen and react to the message.

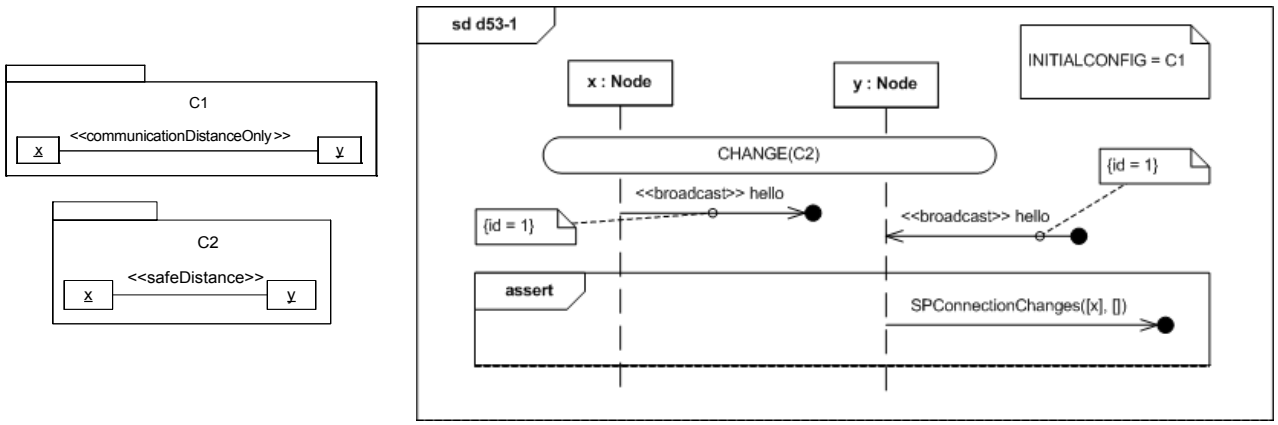
- *Context awareness*. Each node should have an explicit definition of context, of policies to update the context and to react to contextual changes. The context includes any detectable and relevant attribute of a device, of its interaction with other devices and of its surrounding environment at an instant of time. For example, the context can be information collected by means of physical sensors such as location, time, speed of vehicle, or it can be information about network parameters, such as bandwidth, delay and connection topology. Due to mobility, the context is continuously evolving, so that mobile applications have to be aware of, and adapt to, the induced changes.

Existing scenario languages do not offer concepts to account for the dynamically changing structure and context, nor do they offer concepts to represent broadcast communication in local vicinity. We proposed extensions to fill these gaps, and integrated them into a widely used scenario language, namely UML 2.0 Sequence Diagrams [UMLsup]. The extended Sequence Diagrams include two connected views, the spatial view (describing the topological configurations of the system nodes, as well as some contextual information) and the event view (describing communication events, and their causal dependencies on configuration change events). More precisely:

- The spatial view consists of a set of labelled graphs, corresponding to the various configurations that occur in the scenario. For a given configuration, the labels attached to vertices and edges represent relevant attributes of system nodes and of communication links between nodes.
- The event view makes it explicit which communication event occurs in which spatial configuration, and configuration changes are introduced as global events.
- Broadcast communication in local vicinity is introduced by means of special symbols.

Figure 48 exemplifies how we defined the three above extensions in terms of UML elements. The figure represents a simple requirement scenario from a testing case study we investigated, a partitionable Group Membership Protocol (GMP) in the ad-hoc domain. In this protocol [Hua], groups split and merge according to location information carried by *hello* messages. Decision is based on the notion of safe distance, where the safe distance is strictly lower than communication range. The requirement says that whenever a node detects a new neighbour at a safe distance, it has to report the connection change. In the event view, note the global configuration change event, as well as the <<*broadcast*>> stereotype attached to the *hello* message. In the spatial view, it is the responsibility of the designer to determine convenient abstractions for the concrete configurations, depending on the target application. Here, the GMP behaviour is governed by two relations, being at communication range and being at a safe distance. This explains the chosen edge labels. Nodes are merely characterized by their id (see label variables x and y), but tuple of labels are allowed for applications needing a richer representation of node attributes.

While the three proposed mobility-related extensions are relevant whatever the role of the scenario (requirements scenario, test purpose, or test case), the various roles may involve different profiles for the core UML constructs. For example, the UML 2.0 Testing profile [UML TP] is an example of profile defined for test cases. It differs from Modal Sequence Diagrams [HaMa], another UML-based language that targets requirements scenarios. In order to concretely illustrate the usage of our extensions, we focused on requirements scenarios for which we defined a language (TEst Requirement language for MObile Setting, TERMOS) that is inspired from Modal Sequence Diagrams and other similar languages.



(a) Spatial view **(b) Event view**

Figure 48: Example of requirement scenario

3.1.3 Automated treatment of scenario descriptions

In our testing framework (see Figure 47), scenario descriptions are not just for documentation. They are intended to be compiled into programs that automatically analyze execution traces. The execution traces are collected on a test platform composed of three categories of facilities[4] (see Figure 49):

- The *context controller* manages the relative position of nodes according to some mobility model and produces contextual data (e.g., location-based data) needed by the application.
- The *application execution support* emulates the executive support for the application code running on nodes.
- The *network simulator* is responsible for simulating the full functionality of a real wireless network. It uses data from the context controller to control the delivery of messages based on the context (e.g., radio communication in a local vicinity).

Note that existing tools may offer facilities that span several categories. For example, the topology emulator developed within HIDENETS may play the role as a network simulator with parts of a context controller and application support (see [D6.3], Section 6.4).

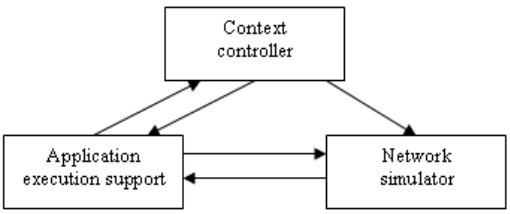


Figure 49: High-level view of the test platform

Basically, the data recorded by the context controller allow us to identify the concrete spatial configurations of the tested system, and the configuration change events. This requires an abstraction step to interpret the raw data in terms of labeled graphs that are then compared to the

[4] A more detailed discussion of test platforms can be found in Deliverable D5.2.

configurations of the scenario. Communication events from the event view are observed by proper instrumentation of the network simulator and application execution support.

The comparison of scenarios and traces serve different objectives, depending on the role of the scenario. *Requirements scenarios* are used to check whether key properties are violated during testing. This offers an automated solution to the test oracle problem, namely how to determine acceptance or rejection of the test outputs. *Test purposes* are used to check whether desired fragments of behaviour are covered at least once during testing. *Test cases* need concrete contextual data (e.g., GPS coordinates) to implement the desired evolution of configurations, and the proposed solution is to extract matching data from preliminary runs of the context controller. The found matches provide concrete data that may be replayed to control the execution of the test case.

In each case, the identified treatments involve graph matching problems, at least for some part. This is due to the need to determine whether the physical nodes appearing in the trace can match abstract nodes appearing in the spatial view. We have developed a graph matching tool, *GraphSeq* (Graph matching tool for Sequences of configurations), to fulfil this need. Determining whether one graph G1 (here, coming from an abstract scenario) is matched by a subgraph of G2 (coming from a trace) can be solved by graph homomorphism building, which has been extensively studied in the literature. Here, the novelty of our tool consists in reasoning on sequences of graphs (i.e., sequences of spatial configurations). It compounds the matching problem, because of the need to retain consistent valuation choices across the sequences of matches. Specifically, the accounting for abstract scenarios where nodes dynamically appear and disappear proved a tricky issue.

Once graph matching has determined which physical nodes can play the role of the nodes appearing in the scenario, trace analysis can proceed by comparing their communication events with the ones in the event view. This requires a well-defined semantics for the event view. We investigated this issue for TERMOS. As a general comment, the problems we encountered did not originate from the language extensions we proposed (broadcast communication, causal dependency on configuration change events). Rather, they came from the core UML constructs. An overview of UML 2.0 Sequence Diagrams semantic problems can be found in [MiWae]. The semantics we retained avoids some of these problems by syntactic restrictions in TERMOS (e.g., we do not allow nesting for some language operators). We also made choices that depart from the standard (informal) interpretation of sequence diagrams, e.g., weak sequencing is no longer the default composition operator for language constructs[5]. These restrictions and choices make it possible to assign a clear and unambiguous meaning to the diagrams. The semantics is then defined in a constructive way, by transforming the sequence diagrams into automata capturing the partial order of events. Note that, at the date of the writing of this deliverable, the transformation is specified but not yet implemented.

3.1.4 Overview of the next sections

After the high-level presentation of the testing contribution within HIDENETS, the next two sections provide a more detailed account of the underlying technology.

Section 3.2 presents TERMOS, the scenario language we developed to describe requirements for mobile-based applications. The language includes the three extensions we proposed to account for interactions in mobile settings: introduction of a spatial view to describe system configurations, representation of configuration changes as global events, and representation of broadcast communication (e.g. for radio communication in a local vicinity). TERMOS provides a concrete illustration of how these extensions, defined in terms of UML elements, can be used into a specialized language profile. TERMOS also exemplifies one of the roles for scenarios in the

[5] This choice is non standard but not uncommon for existing semantics, as will be explained in Section 3.2.2.3.

proposed testing framework (see Figure 47), namely the role of requirement scenarios. Requirement scenarios allow an automation of the test oracle procedure. Their semantics is defined by transforming diagrams into automata that categorize traces as valid or invalid. When analyzing test traces against them, any violation of a requirement will yield a Fail verdict. The presentation of TERMOS covers both the syntax and the semantics, hence allowing an illustration of all the above issues.

Section 3.3 presents GraphSeq, a graph matching tool developed to process the spatial view of scenarios. Note that GraphSeq is independent from the details of the UML profile used in the event view. Hence, it could be used in relation with TERMOS, but also in relation with other scenario languages including a spatial view and having global configuration change events in the event view. The tool takes as inputs two sequences of labelled graphs, intended to come respectively from a scenario and a trace, and generates the set of all possible matchings. Section 3.3 describes the algorithms implemented in GraphSeq. They use a facility for graph homomorphism building, which has been taken from another graph tool developed at LAAS-CNRS [Gue]. GraphSeq has been validated by using hundreds of randomly generated sequences of graphs. We also performed experiments using GraphSeq to analyze data traces supplied by a mobility simulator developed at the University of South California [Bai].

3.2 TERMOS: a scenario language for testing requirements

TERMOS is based on UML 2.0 Sequence Diagrams, which are briefly presented in Section 3.2.1. The original Sequence Diagrams specification was modified in the following way: (i) extensions were added that help describing mobile settings, (ii) the usage of some of the core UML elements were restricted to make the checking of requirements feasible, (iii) the interpretation of some of the elements was modified to overcome some problematic situations. Section 3.2.2 details the rationale behind our modifications. Section 3.2.3 presents the syntax of the language, and Section 3.2.4 exemplifies it by means of requirements scenarios extracted from case studies. Section 3.2.5 describes the semantics.

3.2.1 UML 2.0 Sequence Diagrams

Scenarios in UML are modelled with **Interactions**. A Sequence Diagram is a concrete notation to depict *Interactions*. Figure 50 illustrates a basic Interaction. *Lifelines* represent the individual participants in the Interaction, which communicate via *Messages*.

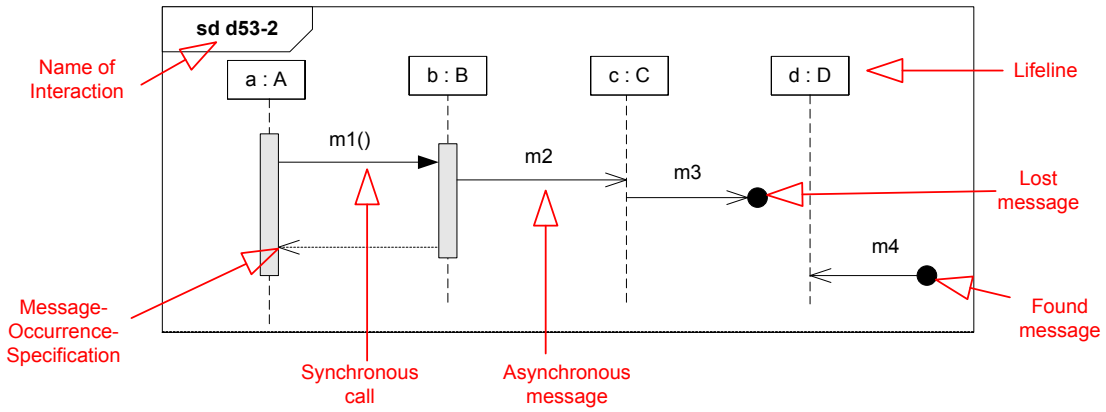


Figure 50: Example Sequence Diagram

Message is a general term; it can be a synchronous or an asynchronous communication. *MessageKind* defines whether the sender or receiver of the message is known (complete, lost or found messages). Messages have two *MessageEnd*s. *OccurrenceSpecification* (and its descendants) is the basic unit of semantics. Sending and receiving messages are marked with *MessageOccurrenceSpecification*.

More complex Interactions can be created with *CombinedFragment* as shown in Figure 51. A *CombinedFragment* consists of one or more *InteractionOperand*s. An *InteractionOperatorKind* specifies the purpose of the fragment. In Figure 51, there is an *alt* operator (i.e., alternative fragment). *InteractionConstraint*s can guard each *InteractionOperand*. Messages on their own cannot cross the boundaries of *CombinedFragment*s, they need a *Gate* which links the two parts of the message. An *InteractionUse* refers to another Interaction. It can pass parameters and can have a return value.

StateInvariant is a runtime constraint on one of the participants of the Interaction. *StateInvariant*s have two kinds of notation, on one hand it can be an expression of attributes and variables, or it can refer to a state of the lifeline's instance (both notations are used on Figure 51). Further constructs exist, for a complete list see the OMG specification [UMLsup].

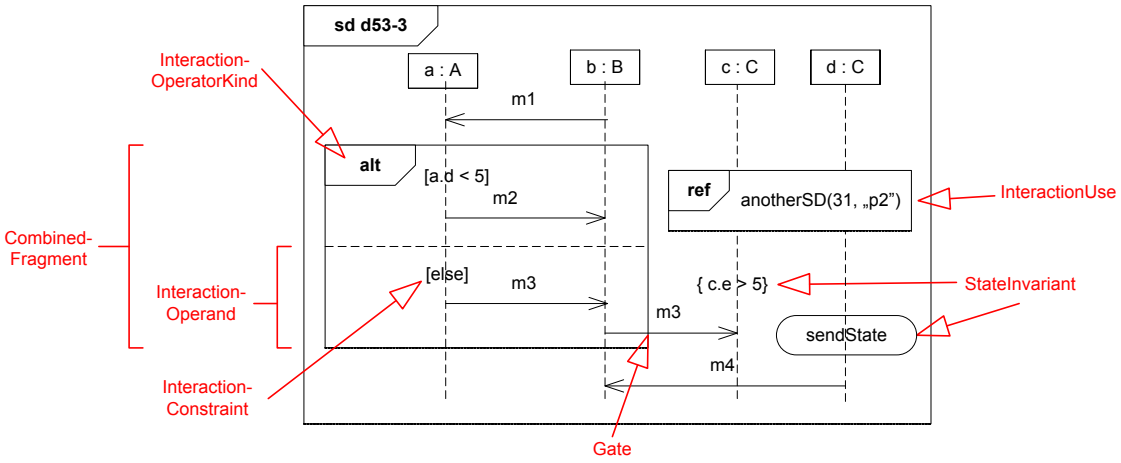


Figure 51: Example for CombinedFragment

Table 1 lists the operators that can be used in *CombinedFragment*s. We grouped the operators in four major categories.

Operators that make the representation of diagrams more compact	
alt	*"alt designates that the CombinedFragment represents a choice of behaviour."*
opt	*"opt designates that the CombinedFragment represents a choice of behaviour where either the (sole) operand happens or nothing happens."*
break	*"break designates that the CombinedFragment represents a breaking scenario in the sense that the operand is a scenario that is performed instead of the remainder of the enclosing InteractionFragment."*
loop	*"loop designates that the CombinedFragment represents a loop. The loop operand will be repeated a number of times."*
Operators that modify the partial order of occurrences	
par	*"par designates that the CombinedFragment represents a parallel merge between the behaviours of the operands."*
seq	*"seq designates that the CombinedFragment represents a weak sequencing between the behaviours of the operands."*
strict	*"strict designates that the CombinedFragment represents a strict sequencing between the behaviours of the operands."*
critical	*"critical designates that the CombinedFragment represents a critical region. A critical region means that the traces of the region cannot be interleaved by other OccurrenceSpecifications."*
Operators that modify the conformance relation	
neg	*"neg designates that the CombinedFragment represents traces that are defined to be invalid."*
assert	*"assert designates that the CombinedFragment represents an assertion. The sequences of the operand of the assertion are the only valid continuations. All other continuations result in an invalid trace."*
ignore	*"ignore designates that there are some message types that are not shown within this combined fragment."*
consider	*"consider designates which messages should be considered within this combined fragment."*

Table 1: Operators in *CombinedFragment*

3.2.2 Discussion of the design decisions for TERMOS

A TERMOS scenario can be seen as describing an observer to check invariant properties. It is preferable to keep it simple, i.e. a "big" property should be decomposed into a set of smaller ones. We do not allow hierarchical description of requirements with sequential composition of scenarios, or references. A set of scenarios corresponds to a set of independent, self-contained checks. The semantics should make it possible to unambiguously determine whether or not a test trace violates any one of the required properties.

The 2.0 version of the UML specification vastly expanded the capabilities of Sequence Diagrams, many elements were incorporated from Message Sequence Charts [MSC], and the semantics was completely redesigned to fit it into the general run-time behaviour of UML. However, a precise semantics was not defined for all of the new elements, which could result in hard to interpret diagrams (see e.g., the discussion we conducted in [MiWae]). In this section, we review some of the potential problems and justify the interpretation choices and syntactic restrictions we retained.

3.2.2.1 Default interpretation of diagrams

Let us start with a simple diagram containing a basic interaction (Figure 52). What does it say about the traces of some target system? As there is no *assert* or *neg* operators introducing mandatory or forbidden modalities, the usual interpretation is that the diagram represents a potential behaviour,

that is, it shows an example of valid trace. For some of the semantics [Stö, Küs], the valid trace is exactly: "!m1,?m1,!m2,?m2" (where '!' denotes sending, '?' denotes receiving) and all other traces are inconclusive. However, when it comes to representing requirements scenarios, it is more convenient to adopt a different interpretation:

- The scenario represents a behaviour fragment, that is, there may be a prefix in the system trace before the shown behaviour occurs, and the behaviour may occur several times in a given trace.
- Also, not all lifelines and not all interactions are represented. The shown events may interleave with other events that are not explicitly mentioned in the diagram.

The second item raises the issue of the identification of events that are allowed to, or not allowed to, interleave. Again, several interpretations are possible (see the discussion conducted in [Klo]). The *strict interpretation* is that the diagram is complete with respect to occurrence specifications that are given in it explicitly. In particular, duplicate messages are not allowed: for a trace fragment "…,!m1,?m1,!m1,?m1,!m2,?m2, …", the first m1 message does not match the m1 message represented in Figure 52, while the second one does. Interleaving with other messages is always allowed (e.g., interleaving with a hypothetical message m3), because a message that does not appear explicitly in the requirement scenario is assumed irrelevant to its trace-language and its satisfaction relation. The *weak interpretation* is less restrictive with respect to the shown occurrence specifications. It only requires that the trace events occur in the specified order (e.g., the m2 interaction is in the future of m1) and may as well accept duplicates.

We have retained Klose's weak interpretation. The UML *consider* operator may then be used to restrict the allowed interleavings (e.g., in Figure 52, considering m1 yields the strict interpretation with no duplicate). The *ignore* operator is not needed, because the default interpretation already "ignores" every event that may interleave with the represented ones.

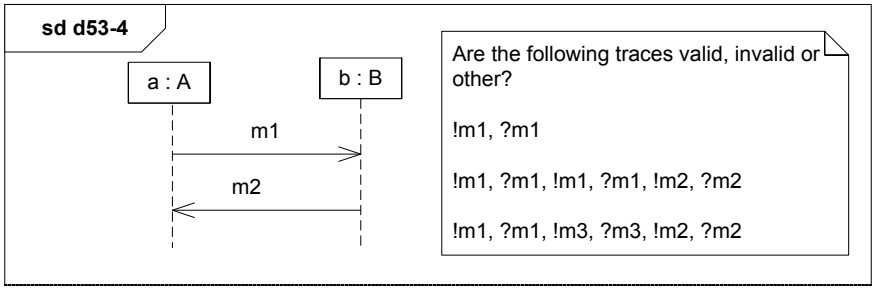


Figure 52: Interpretation of a basic Interaction

3.2.2.2 What is a trace?

Since the purpose of the semantics of the language is to categorize test traces as valid or invalid, a definition of traces is needed. In Figure 52, for the sake of simplicity, a trace was noted as a sequence of message sending (denoted by !m) and receiving (denoted by ?m) events. However, this is not sufficient. Some of the semantics [Stö, UMLsup] actually consider a tuple (!m, sender) or (?m, receiver) for denoting events, while others use the tuple (!m, sender, receiver) However, this is also not always sufficient. Consider for example a trace, where the same event appears between two nodes several times:

"…, (!m1, a, b), (!m1, a, b), (?m1, a, b), (?m1, a, b), …".

Does the first receiving event correspond to the first, or second sending event?

In order to analyze traces, we need to match receiving events with the sending event that caused it, which is only possible, if each message can be uniquely identified. See e.g. [Hal] for such a definition of a trace, and for applications to monitoring distributed systems. Accordingly, in our semantics, a concrete trace will be a tuple containing (?m, receiver, id) or (!m, sender, id), where *m* is the name of the message sent or received, and *id* is an identifier generated by the monitoring functions of the test platform. Note, that a send event may be aimed to several receivers (e.g., in the case of a broadcast message), but a receive event involves only one receiver. The id serves the purpose to match the sending and receiving events of a given message.

When comparing a concrete trace to a requirement scenario, we need to account for the *consider* operators that restrict the allowed interleavings. A problem is that in standard UML, the granularity of *consider* is the message, not the event. How should then a *CombinedFragment* "*consider {m}"* be interpreted? Does it mean that none of the involved lifelines may send or receive messages with type *m* others that the ones explicitly depicted inside the consider fragment? Or do we allow receiving provided that the corresponding send event was not forbidden (for example, consider a hello message sent by a node that is not involved in the scenario)? Moreover, the message granularity limits the expressiveness of the language. In the requirements we analyzed, it was sufficient to use consider in the scope of all lifelines, thus we settled with the following interpretation for *consider {m}*. The sending of messages with type *m* is forbidden for all lifelines, but it is allowed to receive a message with type *m*, if its sending was not forbidden.

3.2.2.3 Synchronization on entering and exiting fragments

The OMG specification uses weak sequencing as the default composing operator between messages and fragments. This means that there is no synchronization mechanism amongst lifelines when entering or exiting fragments.

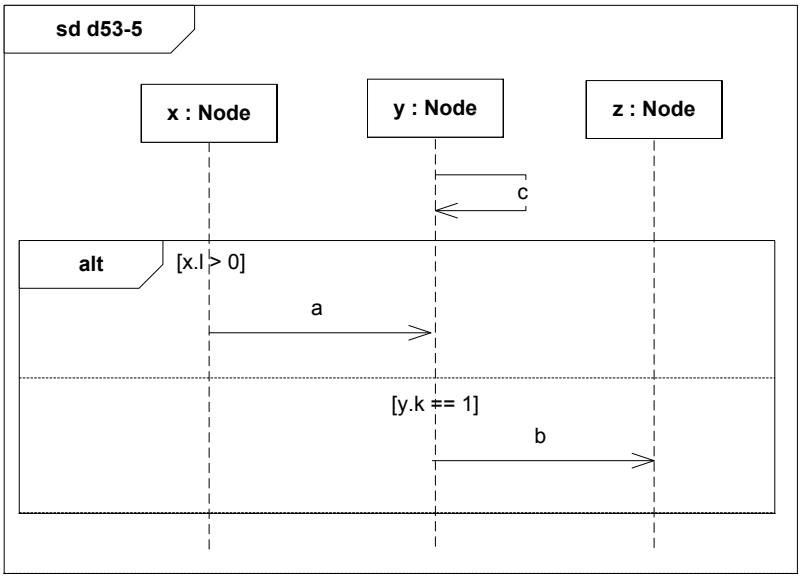


Figure 53: No synchronization on entering fragments

In our testing framework for mobile distributed systems these kinds of diagrams could cause issues in several ways:

- Operators having different meaning than in structured programming languages: E.g., there is no strict sequencing between the iterations of a loop.

- No common point of time to evaluate guards: According to the OMG specification, guards should be placed on the lifeline, which has the first event in that fragment6. Hence, the specification allows non-local choices. This could be problematic in the case of test requirements, because no common point of time could be selected to evaluate the different guards. E.g., on Figure 53, checking of y's guard can happen only after the receiving of message *c*, while checking x's guard can be done before even sending *c*.
- Unclear spatial or temporal scope of conformance operators: Instances can ever enter independently into fragments with conformance operators (*assert* or *negate*, and *consider* or *ignore*). This complicates the checking of requirements heavily, consider for example a scenario where only one instance has entered an ignore fragment so far, while the others are still outside of it.

Thus, we depart from the OMG specification's semantics, and introduce an explicit synchronization on entering and exiting combined fragments among the lifelines covered by the fragment. This is also not an uncommon design decision, see e.g. [HaMa] or [CavFil].

3.2.2.4 Restriction on guards and state invariants

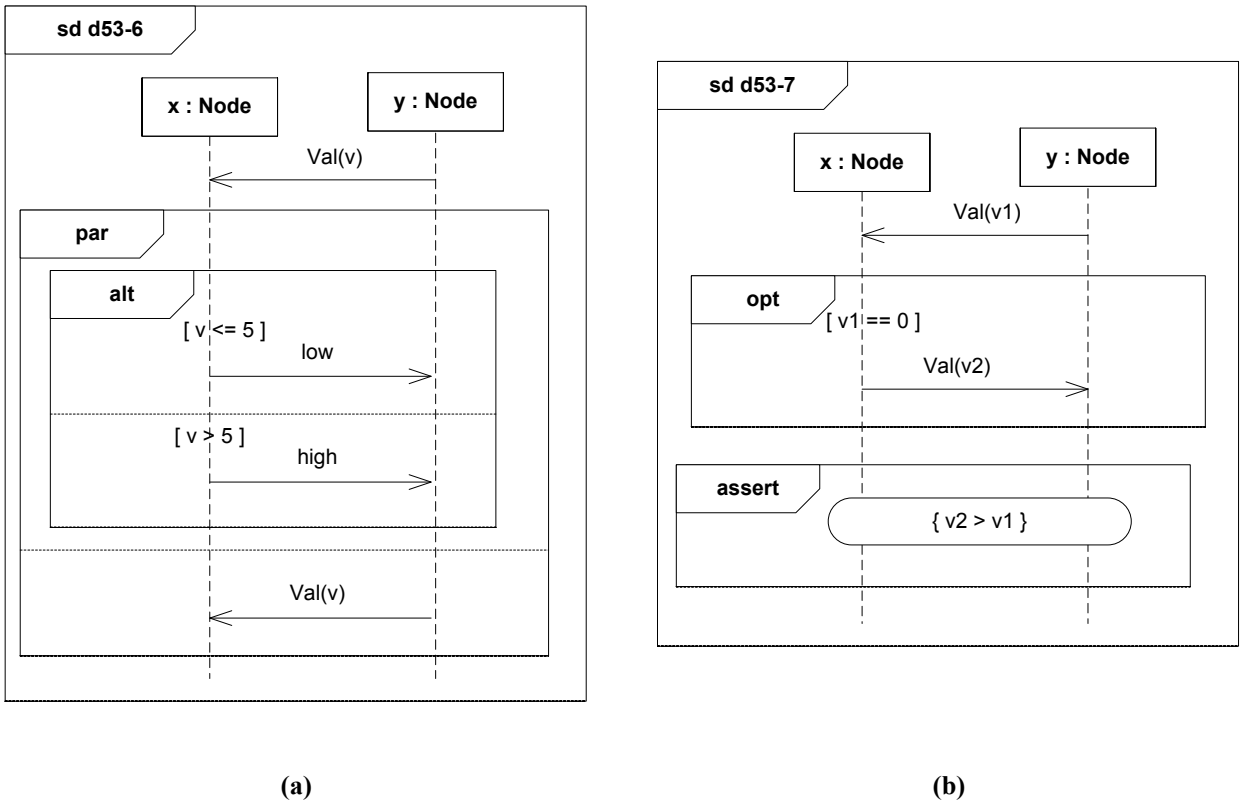
In Figure 53, the synchronization on entering the *alt* box means that the guards must be evaluated after the receiving of *c*. Since requirement scenarios describe partial behaviour, there may be other communication events interleaving with the represented ones. Specifically, there may be events occurring between the receiving of *c*, and the sending of either *a* or *b*. What about then if such events modify the truth value of the guards? Similarly, unrepresented events may have a side effect on the truth value of *StateInvariant* elements. Several strategies may be considered for deciding when to evaluate predicates (see e.g., [Klo]), ranging from evaluation as soon as possible to evaluation at an arbitrary instant. To avoid these problems, and have no side effect from unrepresented events, we impose the following constraint on variables appearing in guards and state invariants:

- Local predicates can only refer to (i) parameters of messages previously sent or received by this lifeline, (ii) node label variables for the target lifeline in the current spatial configuration;
- Global predicates can only refer to (i) parameters of messages previously sent or received by any of the involved lifelines, (ii) node label variables for any of the involved lifelines in the current or previous spatial configurations.

The spatial view will be described in Section 3.2.3.1. As will be seen, the description of spatial configurations accommodates label variables to represent node attributes. These variables are assigned a value when a concrete match of the configuration is found in the trace. The assigned values have to remain stable for the duration of the matching. Message parameters are processed similarly to configuration parameters: they are symbolic constants, the value of which is determined by matching. In Figure 54.a, when a concrete trace event matches the first *Val(v)* emission, it determines the value of *v* for the rest of the scenario. In particular, a concrete *Val* message will not match the second operand of *par* if *v* has not the expected value (in the absence of a consider fragment, this means that inadequate *Val* messages are simply ignored). Specifiers should use different variable names whenever they do not intend a unique concrete value.

[6] Note, that because of constructs like *alt* or *par*, there can be cases, where more than one possible next element exists for an event. Thus, sometimes it is not even obvious where to place the guard. In the semantics part (and more specifically Section 3.2.5.1), we will see that this problem does not occur for TERMOS scenarios.

Note that a clean treatment of predicates at the semantic level should involve checks for well-definedness. Figure 54.b illustrates an ill-definedness problem: if *v1* is not zero, the value of *v2* is undefined. How to detect such problems is presented in Section 3.2.5.3.



(a) **(b)**

Figure 54: Variables in guards and state invariants

3.2.2.5 Deterministic diagrams

As illustrated on Figure 55, ambiguous diagrams can be constructed, where a given trace can be both valid and invalid.

An *alt* operator without guards indicates a non-deterministic choice between its operands. The result of this non-deterministic choice in the current diagram is that if an *a* message appears in the trace, it can be either a valid message (depicted in the second operand) or an invalid one (caused by the *neg* operator in the first operand). Furthermore, because the specification of Sequence Diagrams prescribes weak sequencing as the default composition operator, on first sight the orderings in the diagram might not be obvious. The sending of message *c* is not related to the *a* messages, thus, although it appears visually after the *a* messages, and after the *alt* fragment, actually it can be sent before them.

The OMG specification contains several such other cases. If Sequence Diagrams are just used as a high level overview of the system or as a draft specification, these might not cause problems. But in the HIDENETS testing framework, scenarios should serve as a specification for test oracle checks, thus such non-obvious cases must be handled.

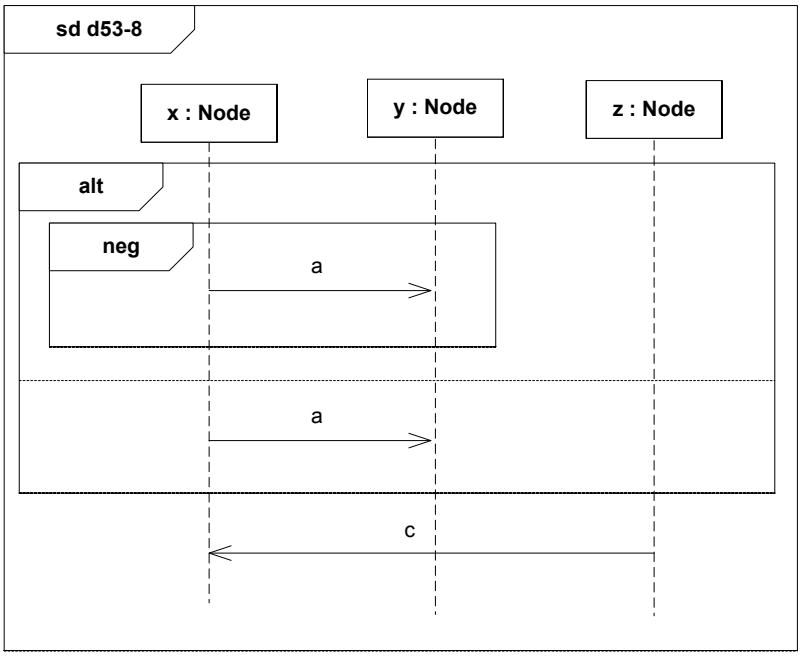


Figure 55: Hard to interpret Sequence Diagram

Ambiguous cases are caused mostly by:

- Unclear scope of conformance operators (when do we start to forbid *a*),
- Non-deterministic constructs in the diagrams (*alt* with several guards true).

This is a problem when a given trace is checked, whether it satisfies a test requirement or not. Thus, we tried to restrict the syntax of the language to make the diagrams deterministic:

- Introducing synchronization on entering and exiting fragments assigns a clear scope to conformance operators in fragments.
- *Alt* is transformed to a deterministic if-then-else construct.

3.2.2.6 No nesting of conformance operators

The OMG specification allows the unlimited nesting of different operators. This could result in an *assert* nested in a *neg* fragment, or in a *par*, where one of the operands contains the negated version of the other operand. Again, checking of test requirements like these is unfeasible, thus we recommended the following modifications:

- The diagram can have only one *assert* box at the end of the diagram, which should cover all lifelines.
- Negative scenarios are expressed with a false global predicate in the assert box instead of a *neg* fragment.
- Only one level of nesting of conformance operators is allowed.
- Table 2 summarizes the allowed combinations of operators.

	alt	opt	par	assert	consider
alt	Y	Y	Y	Y	Y
opt	Y	Y	Y	Y	Y
par	Y	Y	Y	Y	Y
assert	N	N	N	N	Y[1]
consider	N	N	N	Y[2]	N

Table 2: Can the operator in the row be nested in the operator in the column?
[1] The *consider* should be at the main level of the diagram.
[2] The *assert* should be at the main level of the diagram.

3.2.3 Syntax of the language

In the description of the language's syntax, we put emphasis on the new elements we propose to allow description of scenarios in mobile settings. The new elements concern the introduction of a spatial view for the scenario (Section 3.2.3.1), the accounting for spatial configurations in the event view (Section 3.2.3.2), and the representation of broadcast communication (Section 3.2.3.3). We then provide an overview of the syntax of the event view (Section 3.2.3.4), recapitulating the syntactic constraints put on the core UML elements to facilitate the definition of the semantics.

3.2.3.1 Syntax of the spatial view

The spatial view may contain several spatial configurations. Each configuration is given a name, e.g., Figure 56 shows a configuration named C3.

A configuration is a labeled graph, where vertices represent system nodes and edges represent different kinds of connection between nodes. The syntax of the labeled graphs presented below is compatible with the input domain of the graph matching tool described in Section 3.3. Specifically, we put constraints on the number and type of labels, and only consider undirected graphs.

Each node has a symbolic id. For example, Figure 56 shows three nodes having IDs "x", "y" and "z". This means that any scenario referring to C3 must involve lifelines for nodes x, y and z. In order to allow for a richer representation of configurations, nodes can have two additional attributes of integral types (i.e., integers or enumeration types). The corresponding vertex labels in the graph can take different forms:

- A constant value from the integral type. For example, in Figure 56, the two attributes of node y have constant values 1 and 2.
- A variable name, denoting a value from the type. For example, the first attribute of nodes x and z must be identical, but their precise value is let unspecified (variable v1). This value is intended to remain stable in the configuration. Moreover, if a scenario involves several graph configurations containing label variable v1, it must be substituted for a single value. Thus, v1 can be seen as a symbolic global constant for the scenario.
- A wildcard indicating a *don't care* value, see e.g. the second attribute of node x. *Don't care* values do not need to remain stable in the given configuration.

Edges can be labelled by constant values or wildcards. In Figure 56, it is assumed that the connection type is an enumerated type {*safeDistance*, *communicationDistanceOnly*}, like in the GMP testing case study. Nodes x and y have a *safeDistance* connection; Nodes y and z are disconnected; we do not care about the connection of nodes x and z, they may exhibit unstable connections/disconnections during the configuration.

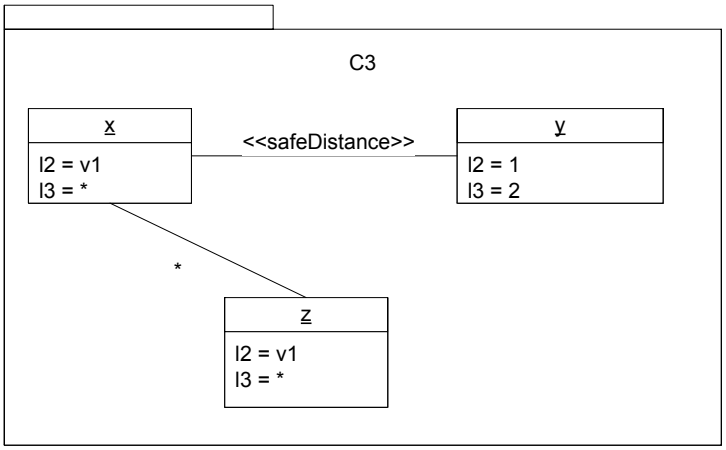


Figure 56: Example of spatial configuration

To be as compatible with the original UML specification as possible, we depict spatial configurations using object diagrams. A package with the name of the configuration contains all elements. Nodes are represented as instances, slots named *l2* and *l3* contain the additional labels defined for the given node. Labels for edges are represented as stereotypes, because they characterize the given connection between the two nodes.

3.2.3.2 Spatial elements in the event view

The event view of a scenario uses UML 2.0 Sequence Diagrams, with some extensions to explicitly account for the spatial configurations defined in the spatial view.

An Interaction can be tagged with the *termosScenario* stereotype (Figure 57) to show that is a requirement scenario in TERMOS. The *termosScenario* stereotype has an association named *initialConfiguration* giving the initial configuration of the Interaction.

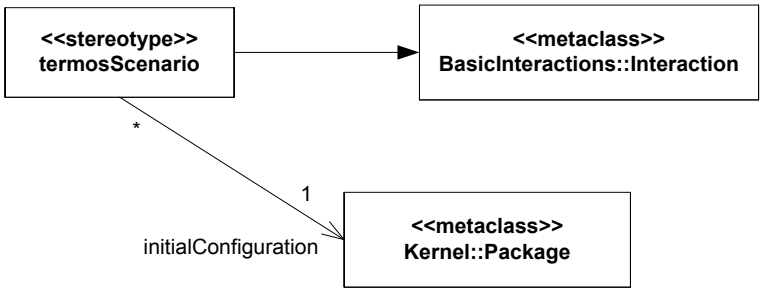


Figure 57: The termosScenario stereotype

This extension fits well into the UML framework, the only drawback is that because Interactions are the abstract concepts representing scenarios, they visually do not appear on a Sequence Diagram. In most of the modelling tools, assigning a stereotype to an Interaction is only reflected in the textual properties view, but not on the diagram itself. Figure 58 illustrates the case, where the *termosScenario* stereotype is assigned to an Interaction named *ReportHello* in Rational Software Architect [RSA]. For this reason, in the examples used in this deliverable we depict the initial configuration of the diagrams in a comment box containing the text *INITIALCONFIG* also. These comments are not part of the semantic model, rather they ease the readability of the examples.

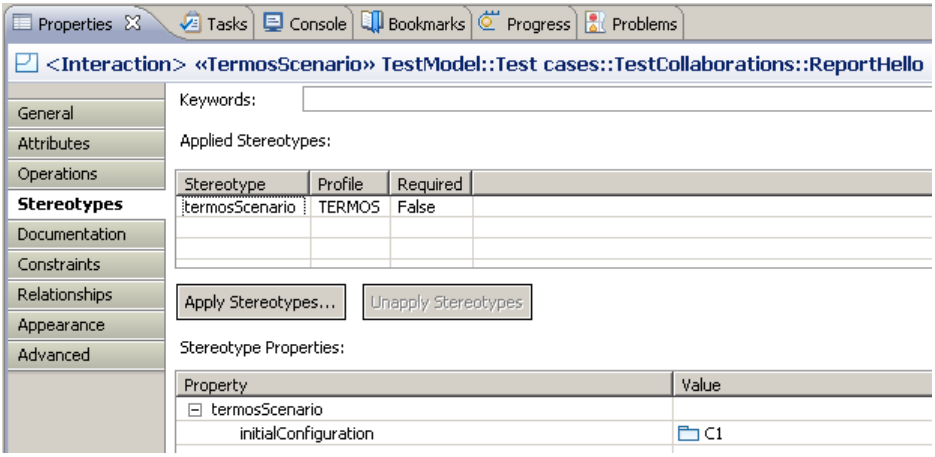


Figure 58: Assigning the termosScenario to an Interaction

Configuration changes are then represented by global events of the form *CHANGE (name_of_new_config*) that induce a global synchronization for all lifelines. Configuration changes cannot be nested into operators, except into a consider operator that is at the main level. In particular, we cannot require a configuration change (nesting into *assert*). Configuration changes are "decided" by the environment. Also, there is nothing such as an optional or parallel configuration change (nesting into *opt*, *alt* and *par*). Configuration changes arise deterministically and involve all lifelines at the same time. In this way, the diagram can be decomposed into fragments, where each fragment takes place in a well-defined spatial configuration. This makes it explicit which communication event occurs in which configuration. Predicates (guards of *alt* operands, state invariants) may refer to variables of their current or past configurations (i.e., node label variables), see Figure 59.

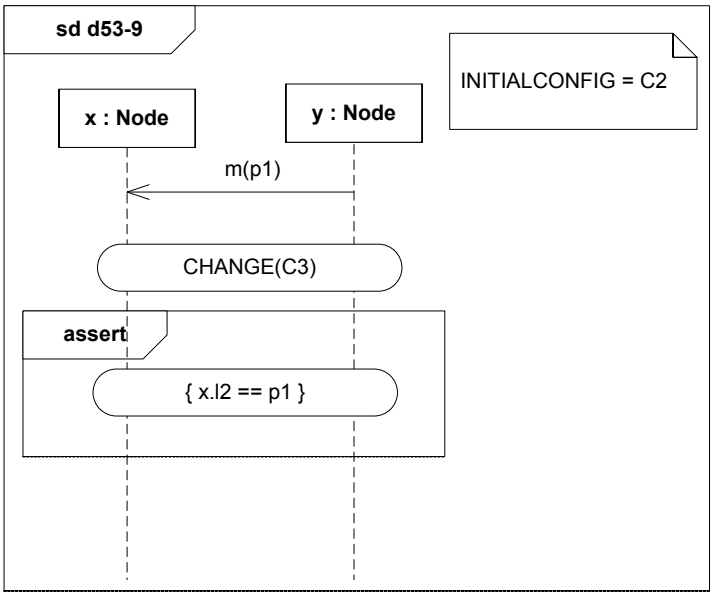


Figure 59: Example for configuration changes

The configuration changes may involve the dynamic creation, shutdown and restart of nodes. For example, a scenario may have three successive configurations C4, C5, C6, where:

- C4 contains a node with id *x*,

- C5 does not contain a node with id *x*, but contains a node with id *y* that was not present in C4,
- C6 contains both nodes *x* and *y*.

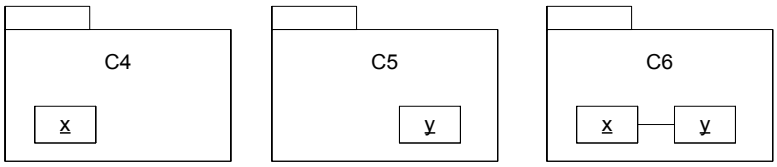


Figure 60: Spatial configurations with node changes

There is no convenient way to describe such a dynamic structure in sequence diagrams. For example, a lifeline can be stopped, but then it is not possible to restart it. Also, dynamic creation can only occur as the result of an action performed by an existing lifeline.

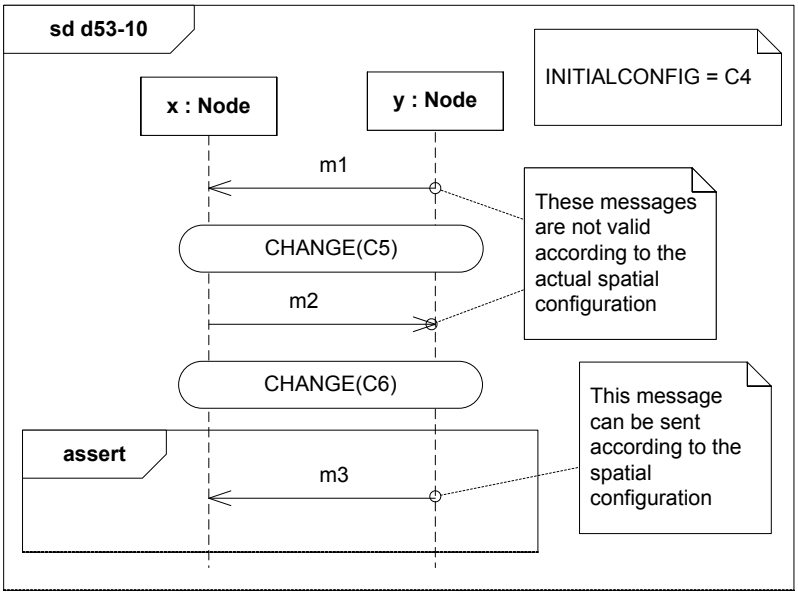


Figure 61: Invalid message according to the spatial configuration

To solve this problem, we take the convention that the spatial configuration determines which node is alive/dead at some point of the scenario. There is a lifeline for every nodes mentioned in any one of the configurations. If a node is not active at some point of the scenario, then it is not supposed to participate to any communication interaction (Figure 61). Checks can be provided to warn the scenario specifier whenever communication is not compatible with the spatial view:

- Dead nodes sending and receiving messages,
- Active nodes exchanging messages while there is no path connecting them in the current configuration.

3.2.3.3 Broadcast communication

UML sequence diagrams focus on point to point communication. There is no element dedicated to the representation of broadcasts or multicasts. This is a serious drawback for representing local broadcasts, i.e., communication with unknown partners in local vicinity.

We propose to use the concepts of lost and found messages to represent such broadcasts. Lost messages are messages with no explicit receiver. Similarly, found messages do not have an explicit sender. Lost and found messages offer flexibility to represent partial behaviour, where not all lifelines and not all communication events are of interest. Such flexibility is quite useful when specifying requirement scenarios; hence we need lost and found messages independently of our consideration for local broadcasts.

In order to distinguish broadcasts from "usual" lost/found messages, we assign them the <<broadcast>> stereotype. A broadcast involves one send event followed by one or several receive events. A tagged value is attached to the corresponding lost/found messages, so that each receive event of the diagram can be paired to the send event that caused it. Figure 62 presents the definition of the broadcast stereotype.

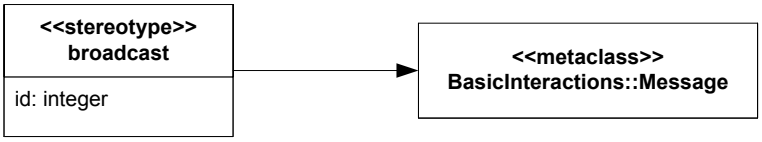


Figure 62: The broadcast stereotype

Figure 63 shows an example how the broadcast stereotype can be used. There are two broadcast messages on the diagram, one send by node *x* (identified by id 1) and one by node *z* (identified by id 2). Every other node receives the broadcasts messages, as depicted by the found messages.

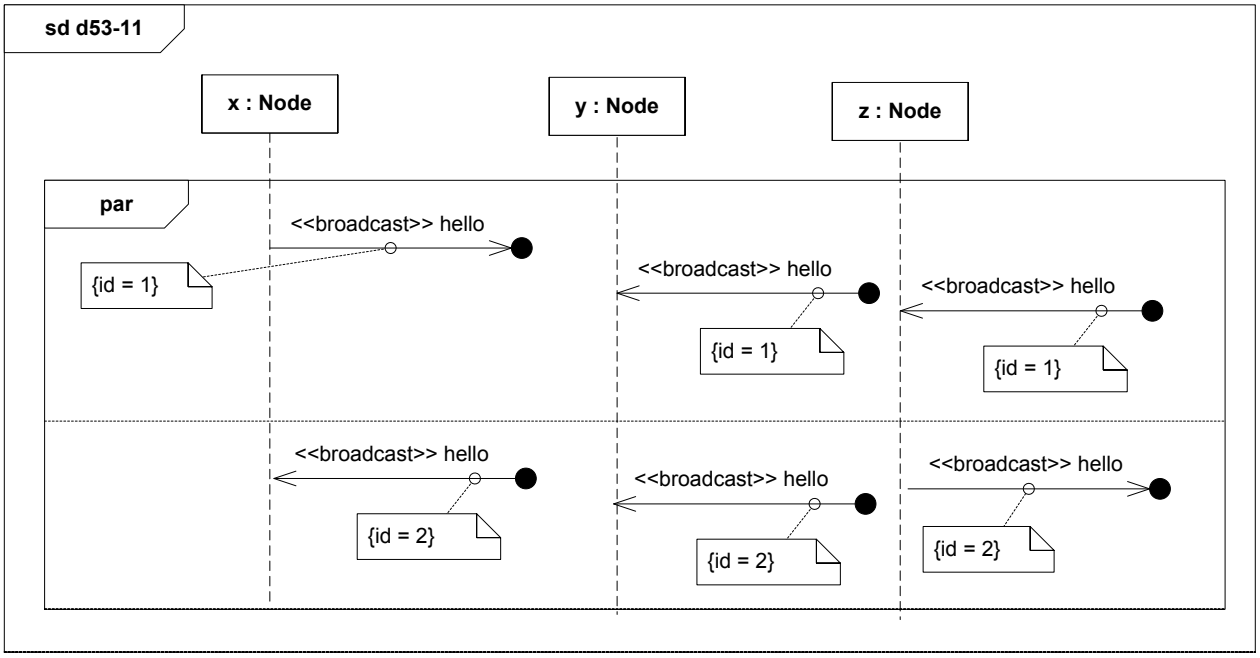


Figure 63: Example of broadcast messages

3.2.3.4 Syntax of event view

The abstract and concrete syntax of the event view are derived from the syntax of UML 2.0 Sequence Diagrams. According to the decisions described in 3.2.2, some of the elements were removed and some additional constraints were added to adopt it our environment.

Appendix B contains the complete abstract syntax of TERMOS. The changes to the original abstract syntax are collected in Table 3.

Type of change	Description of change
Remove	Removed elements: Events, Gate, PartDecomposition, GeneralOrdering, Continuation, ExecutionSpecification.
Remove	The following operators were removed: *seq, strict, loop, ignore, neg, break, critical*.
Change	Changed the multiplicity for the association going from StateInvariant to Lifeline from 1 to 1..* to allow global predicates. The concrete syntax remains the same, just now StateInvariants can span to multiple Lifelines.
Constraint	Only the following operators can have guards: *alt, opt*.
Constraint	The following operators have only one operand: *opt*, *assert*, *consider*.
Constraint	The *assert* and *consider* operators should cover all Lifelines.
Constraint	There should be an *assert* fragment at the bottom of the diagram.
Constraint	If a FALSE global predicate is used, it is the only element in the *assert*, and covers all lifelines.
Constraint	The nesting of conformance operators is only allowed as in Table 2.
Constraint	The configuration change can only be in the main fragment of the diagram or nested in a *consider*, provided that the *consider* is at the main fragment of the diagram.
Constraint	The diagram should contain a note with the initial configuration in it.

Table 3: Changes to the original Sequence Diagram syntax

Automated checks can be implemented to verify that a diagram conforms to the above changes and constraints. A prototype tool was created in the modelling tool IBM Rational Software Architect to show the feasibility of the approach. Rational Software Architect calls such an extensibility tool used for model manipulation as a *pluglet*. The pluglet implemented takes an Interaction as an input, and outputs whether the diagram violates the above constraints. The Interaction is passed in the internal format of the Eclipse UML2 component, which is basically a Java representation of the Interaction stored in its abstract syntax. The following example presents a fragment from an Interaction represented in that format.

```
org.eclipse.uml2.uml.internal.impl.InteractionImpl@f700f70 (name: Interaction1)
  org.eclipse.uml2.uml.internal.impl.LifelineImpl@6fb46fb4 (name: a)
  org.eclipse.uml2.uml.internal.impl.LifelineImpl@71667166 (name: b)
  ...
  org.eclipse.uml2.uml.internal.impl.MessageOccurrenceSpecificationImpl@73f873f8
    (name: <unset>, visibility: <unset>)
  org.eclipse.uml2.uml.internal.impl.MessageOccurrenceSpecificationImpl@75a475a4
    (name: <unset>, visibility: <unset>)
  org.eclipse.uml2.uml.internal.impl.CombinedFragmentImpl@3d803d80 (name:
    first_assert, visibility: <unset>) (interactionOperator: assert)
      org.eclipse.uml2.uml.internal.impl.InteractionOperandImpl@4c864c86 (name:
       <unset>, visibility: <unset>)
        org.eclipse.uml2.uml.internal.impl.InteractionConstraintImpl@179e179e
```

```
(name: <unset>, visibility: <unset>) (visibility: public)
  org.eclipse.uml2.uml.internal.impl.OpaqueExpressionImpl@439a439a
   (name: <unset>) (body: [a.k > 5], language: null)
```

Figure 64 presents the pluglet in action. The tool currently analyzed Interaction1 selected from the tree view on the left side. Interaction1 is depicted on SequenceDiagram2 showed in the center of the screen. Finally, the result of the analysis is presented as console messages in the lower part of the screen. It can be seen that the interaction violates several constraints: there are several assert fragments in the diagram, some of the asserts are not covering all lifelines and they have guards, etc.

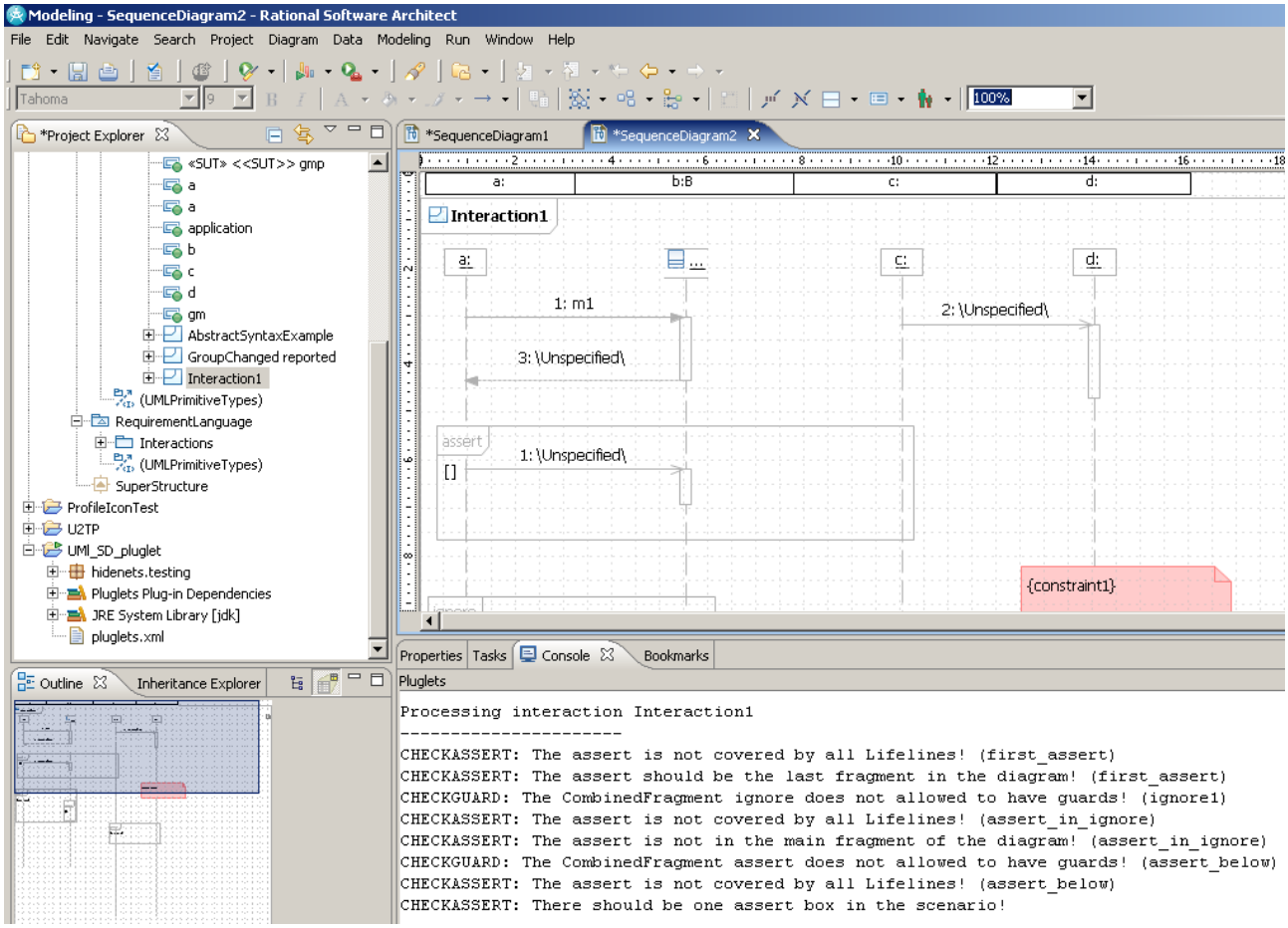


Figure 64: The pluglet checking the extra constraints for the language

Apart from the simple constraints presented in Table 3, there are other, more complex checks that could be done to validate whether a requirement scenario is also semantically well-formed.

- Check, whether messages depicted on the diagram can be sent and received in the current spatial configuration.
- Check, whether predicates refer only to message parameters received so far and to configuration labels from the current or past configurations.

These checks will be addressed later in the semantics part in Section 3.2.5.3.

3.2.4 Example scenarios

The following scenarios exemplify the recommended new elements in the language.

3.2.4.1 Group Membership Protocol

The Group Membership Protocol we analyzed [Waes] provides a consistent view of nearby nodes in mobile ad-hoc environments. Nodes form groups, and each group has a leader. A leader informs the members of its group with *SPGroupChange* messages about a change in the group membership.

Figure 48 and Figure 63 showed already examples for the GMP with broadcast messages and configuration changes. The next examples will present how local and global predicates can be used in requirement scenarios.

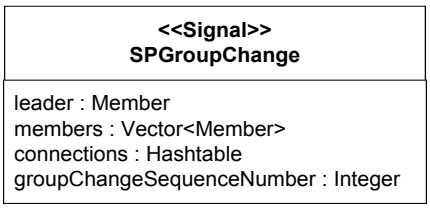


Figure 65: SPGroupChange message

The *SPGroupChange* message contains a *groupChangeSequenceNumber*, which identifies the new group. The following requirements can be defined for this protocol.

Local monotonicity: Group identifiers installed on each host are in increasing order.

This requirement can be captured with the following scenario.

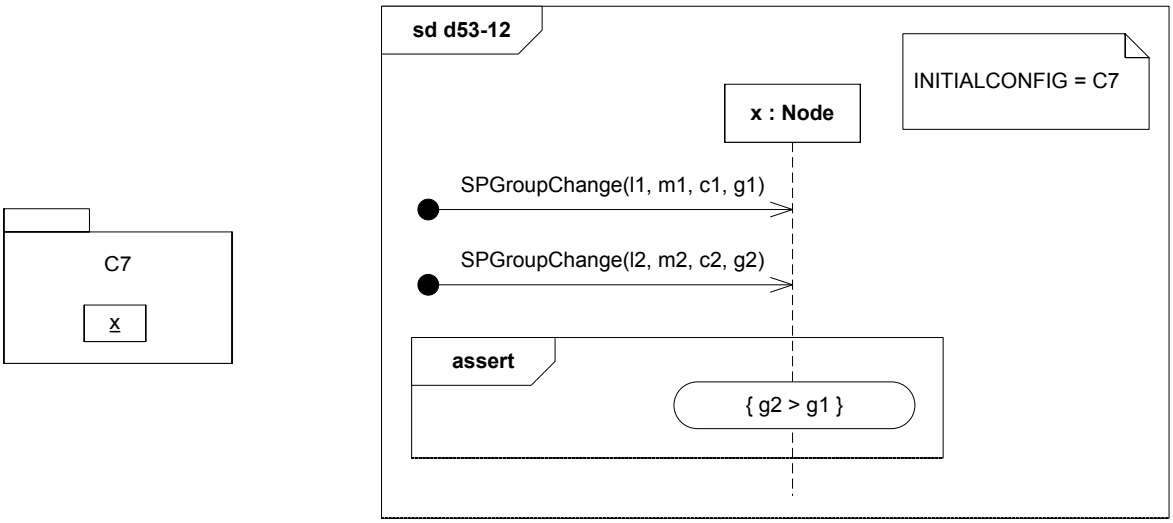


Figure 66: Local monotonicity requirement for the GMP

This example showed how message parameters can be used in state invariants to express predicates.

An additional requirement for the GMP is membership agreement.

Membership Agreement: If hosts x and y have the same group id, then they have the same views.

The initial spatial configuration of the diagram shows that x and y are in safe distance (which is a requirement for being in the same group).

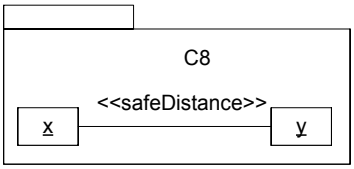


Figure 67: Spatial configuration for membership agreement requirement

The event view of the requirement depicts that if the two nodes receive a group change message for the same group, then these messages should contain the same membership. Note that here a global state invariant is used, i.e. one that refers to values belonging to different nodes.

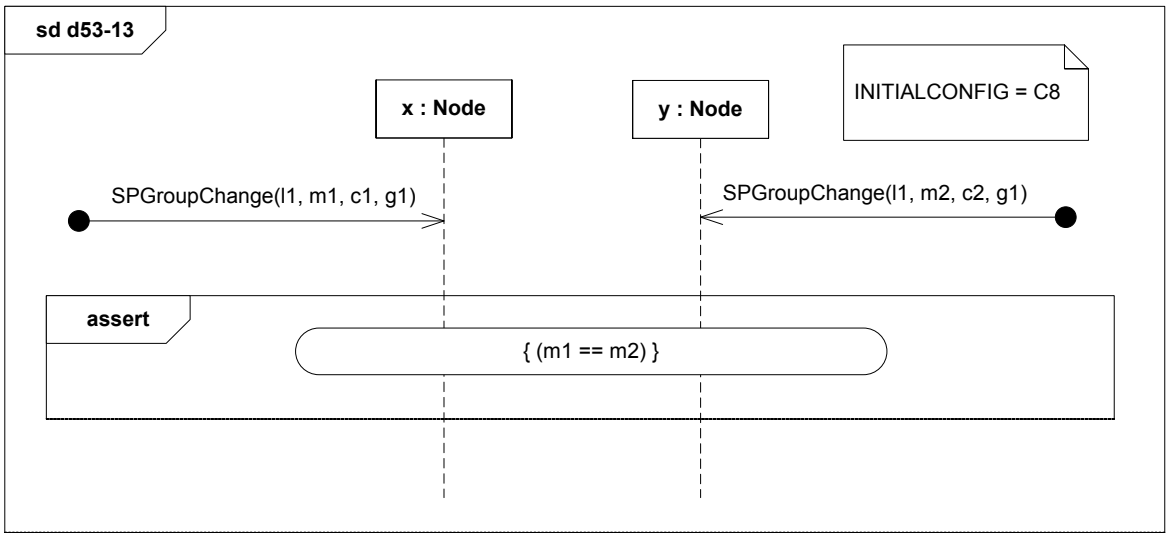


Figure 68: Event view for the membership agreement requirement

3.2.4.2 Platoon Driver Support Software

The Platoon Driver Support Software (PDSS) [D6.3] is one of the demonstrators developed in HIDENETS. It simulates the control of a platoon consisting of a head vehicle and several slave vehicles. The head vehicle collects the speed, acceleration, etc. data from the slaves, and provides them new actuation values in response. One of the requirements can be described as:

The head vehicle has to respond for every speed report with a new speed value.

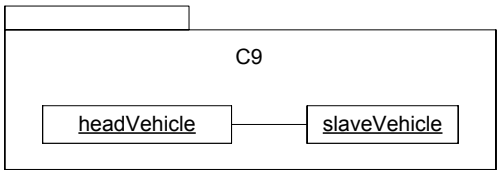


Figure 69: Spatial configuration for the PDSS requirement

The requirement can be depicted with the scenario on Figure 70. The important part of the diagram is the use of a consider fragment. Without the consider fragment, because of the weak interpretation, several reportSpeed messages can appear before sending the setSpeed message. However, the requirement states that the headVehicle has to respond to each report message. With the consider fragment, this behaviour is achieved.

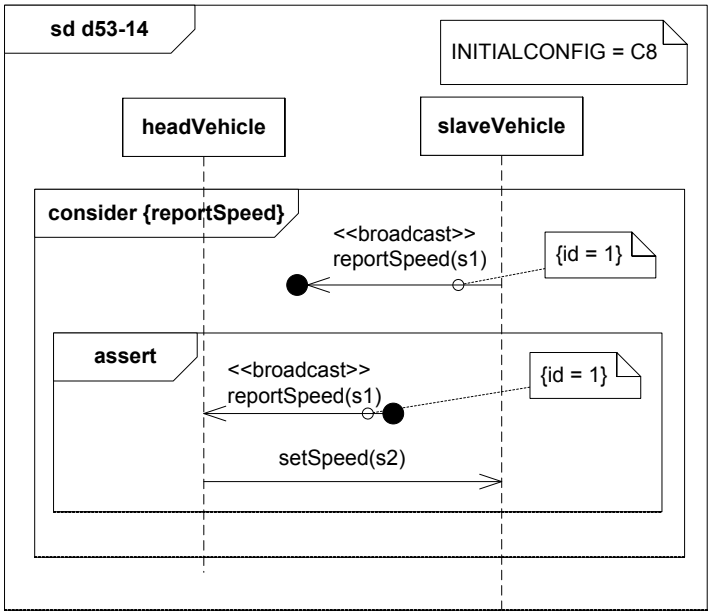


Figure 70: Requirement for the PDSS application

3.2.4.3 Distributed black box application

The distributed black box application [D6.3] in HIDENETS backs up key data from a car to other cars or infrastructure nodes. A requirement defined for this application can be described as follows.

After a car V1 backs up its data on an infrastructure server, it must not back up its data on another car V2.

This is so because the memory space available on neighbouring cars is reserved for data that could not be saved on the infrastructure yet.

The spatial configuration change of this scenario can be expressed as in Figure 71.

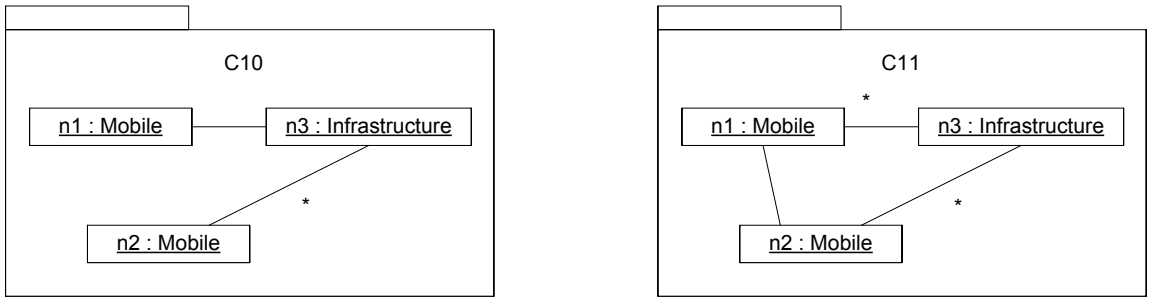


Figure 71: Spatial configurations for the black box application

In the spatial configuration C10, the car n1 is connected to the infrastructure node n3. The car n1 and the car n2 are not connected. The symbol '*' is to express any spatial relation between two nodes.

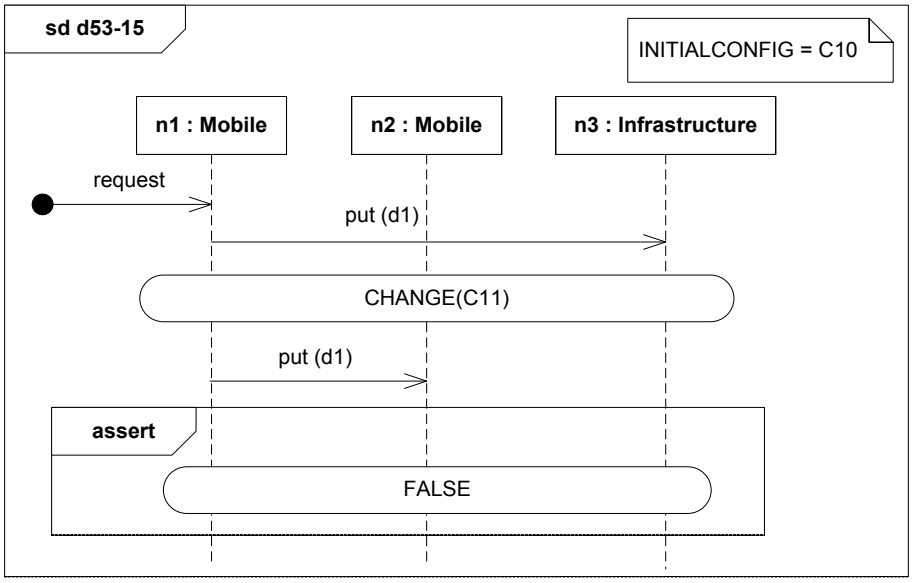


Figure 72: Event view of the black box scenario

Figure 72 depicts the event view of the scenario. The example illustrates how negative scenarios can be expressed using a global, false invariant. A request from the user on n1 asks for the backup of n1's data to the infrastructure. Then, if car n1 is connected to another car (n2) by transmission range (spatial configuration C11), it should not back up the same data on n2, as the data is already on the infrastructure.

3.2.5 Semantics of the language

The semantics of TERMOS has been inspired by the semantics proposed for LSC, more specifically the one defined by Klose [Klo]. The approach builds an automaton from the diagram, the states of the automaton being determined by the valid cuts of the diagram. Informally, a cut is intended to represent a consistent global state characterized by the events occurred so far, and it is meaningful to reason about the past or the future of this state. The automaton transitions then stand for the successor relation among the cuts. Klose's approach has been extended (i) to incorporate UML SD elements not present in LSC, e.g. *alt* or par *combined* fragments, and (ii) to handle the mobile settings related elements, e.g. broadcast messages and configuration changes. Also, the details of the construction of the automaton differ in several aspects:

- Klose builds a Büchi automaton to accommodate infinite traces. Since we are dealing with finite test traces, we are building a standard automaton.
- Klose has a separate treatment for the pre-chart (for us, the analogous would be everything before the *assert* fragment) and chart (for us, would be the content of the *assert* fragment). Our semantics builds a single automaton for the whole diagram.
- We have an interleaving semantics, while Klose allows several events to occur at the same time.

As regards the last two points, our choices are similar to the ones made for a UML variant of LSC, called MSD [HaMa], which also has an interleaving semantics captured in one automaton.

Our definition of the semantics closely follows the steps identified by Klose.

First the diagram is parsed, its basic building blocks and the orderings between them are identified (Section 3.2.5.1). Next, the automaton is constructed using the structures built in the first step

(Section 3.2.5.2). As stated in Section 3.2.3.4 the diagram has to conform also to complex well-formedness rules, which can be checked based on the formal semantics (Section 3.2.5.3). Finally, the automaton built for the event view has to be connected to the spatial view (Section 3.2.5.4). The details of the semantics will be illustrated by taking example scenarios and defining their semantics.

3.2.5.1 Pre-processing the diagram

To create an automaton capturing the semantics of a diagram, first the elements of the diagram are identified.

Definition 1. The basic building block of a TERMOS diagram is called an *atom*. The following elements are atoms:

- Lifeline heads, denoted by $\perp_l$ for Lifeline *l*,
- Lifeline ends, denoted by $\top_l$ for Lifeline *l*,
- *MessageOccurrenceSpecification*s, i.e. sending a message or receiving a message,
- *StateInvariant*s (for global *StateInvariant*s every Lifeline has a separate *StateInvariant* atom),
- configuration changes,
- entering a *CombinedFragment*,
- exiting a *CombinedFragment*,
- guards.

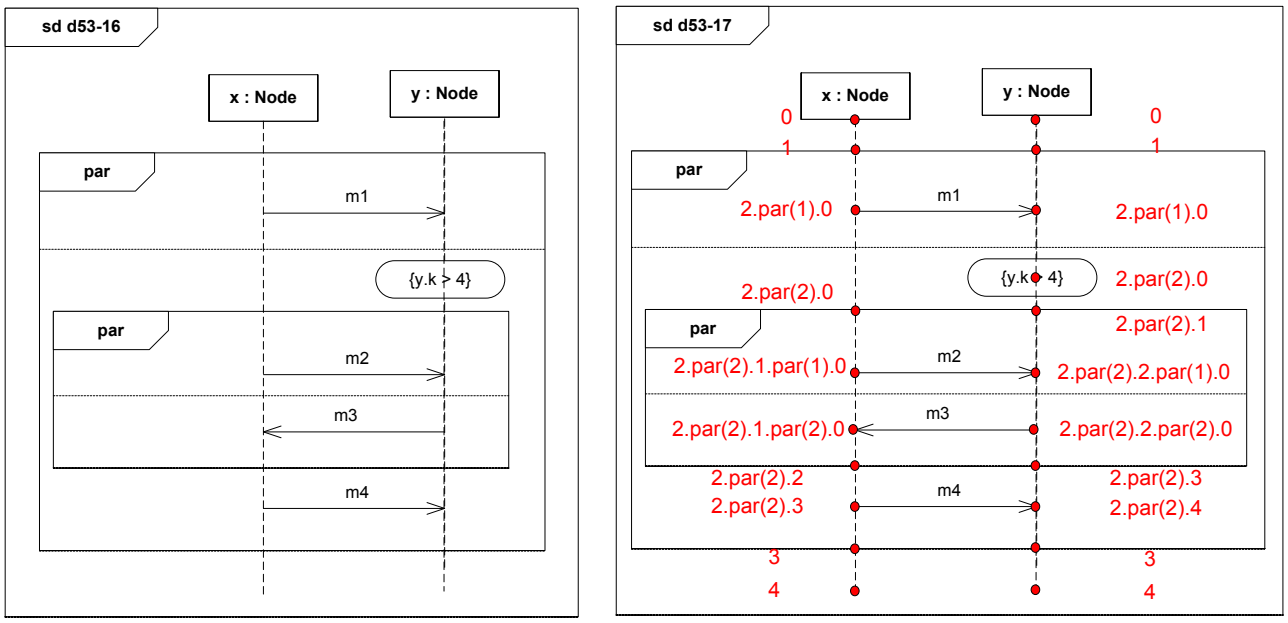


Figure 73: Example for assigning atom position to *par* fragments

The orderings of the atoms on one Lifeline are defined by their *position*. Klose uses an integer as the position of atoms, however this is not sufficient in our case. In the case of parallel or alternate fragments, the visual positioning of atoms does not necessarily mean a temporal relation between them, i.e., the elements inside the second operand of a *par* fragment are drawn below the elements inside the first operand, but they should not necessarily happen after the atoms in the first operand.

To solve this issue, instead of an integer value a path expression is assigned to each atom, similarly to the approach used in [Küs]. The method is illustrated by the following example.

The left side of Figure 73 contains the example diagram, while the right side is annotated with the atom positions. The idea is that for the elements inside the main fragment or for the elements inside one operand, every atom is assigned a number according to their visual position starting from zero. If we enter a *CombinedFragment*, then a path expression is added to the position quantifying in which operand the current atom resides. In the current example this translates to the following positions.

- The head of Lifeline *x* is assigned position 0.
- The atom for entering the first *par* fragment still belongs to the main fragment, thus it gets position 1.
- The *par* gets the next position, which is 2.
- Elements inside the *par* inherit the position of the *par* fragment (namely 2 in the current example), and an expression describing in which operand of the *par* they reside. Thus, sending *m1* on *x* gets 2.par(1).0, meaning that it is in the *par* identified by position 2, it is in the first operand of the *par*, and it is the first atom of that operand.
- Entering the second *par* fragment is in the second operand of the outer *par*, thus it is assigned 2.par(2).0. Sending of *m2* is inside the nested *par*, its position reflects this nesting: 2.par(2).1.par(1).0. The second *par* has the position 2.par(2).1, this position is prefixed to every elements inside that fragment.
- Exiting a fragment belongs to the same level as the fragment itself, thus exiting the first *par* gets the position 3, showing that it is at the main diagram fragment.
- Atom positions are only unique per lifelines, and atoms representing the same event (e.g. entering the same fragment), can have different positions assigned. This is illustrated with the help of positions on lifeline *y*.

Thus the definition of the atom position is the following.

Definition 2. The *atom position* identifies the position of an atom on one lifeline. It has the form *path.id*, where *path* is a string identifying in which CombinedFragment the atom is, and *id* is an integer giving the order of the atom compared to the other atoms inside that fragment. Path is empty if the atom is in the main fragment of the diagram, otherwise it is in the form *p.opr(o)*, where *p* is the position of the CombinedFragment the atom is in, *opr* is the operator of the fragment, and *o* is the number of the operand the atom is in.

The example on Figure 74 shows how atom positions can be assigned to an *alt* fragment.

Guards of operands are grouped to the next atom on the Lifeline, forming a *cluster* with that atom. However, care must be taken, because sometimes there is no next atom inside the guard's operand (e.g., in an empty [else] operand coming from an *opt* fragment). In this case, the cluster contains only the guard. In the original UML specification, there can be several immediate successor of an atom also (e.g., if the atom is right before a *par* with several operands). With the introduction of a separate atom for the beginning of a fragment, this is not the case in TERMOS.

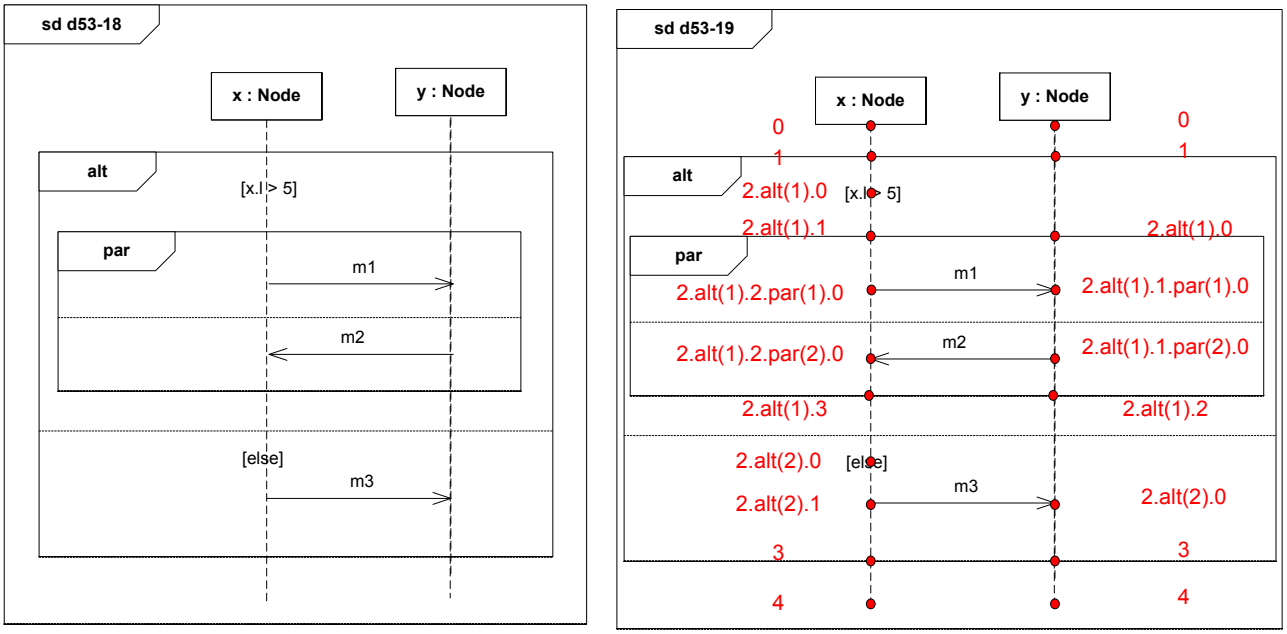


Figure 74. Example for assigning atom position to *alt* fragments

Definition 3. If the *a* atom is a guard with a position *p.i* and there exists an atom with a position *p.(i+1)*, then the two form a cluster. Every other atom forms a cluster with only that atom in it.

To handle the positions of clusters with multiple elements, the concept of location is defined.

Definition 4. The *location(cl)* function returns the minimum of the positions of the atoms inside the cluster, where *min (p.i, p.(i+1)) = p.i*.

Several elements provide synchronization across Lifelines, e.g. configuration changes or entering a fragment, the clusters corresponding to these elements have to be mapped together. Simultaneous classes, *SimClasses*, serve this purpose.

Definition 5. A *simultaneous class*, *SimClass*, is a set of clusters from separate Lifelines. The clusters representing the following elements form a *SimClass* together, every other cluster forms a *SimClass* with only that cluster as its member:

- the beginning of the same *CombinedFragment*,
- the end of the same *CombinedFragment*,
- the same configuration change,
- the same global *StateInvariant*.

Figure 75 illustrates how atoms, clusters and *SimClass*es are defined for a diagram. To sum up: atoms are “points” on lifelines; clusters are used to group simultaneous atoms on a given lifeline; *SimClass*es group clusters that are simultaneous at a diagram-wide level, that is, non singleton *Simclass*es represent synchronization of several lifelines.

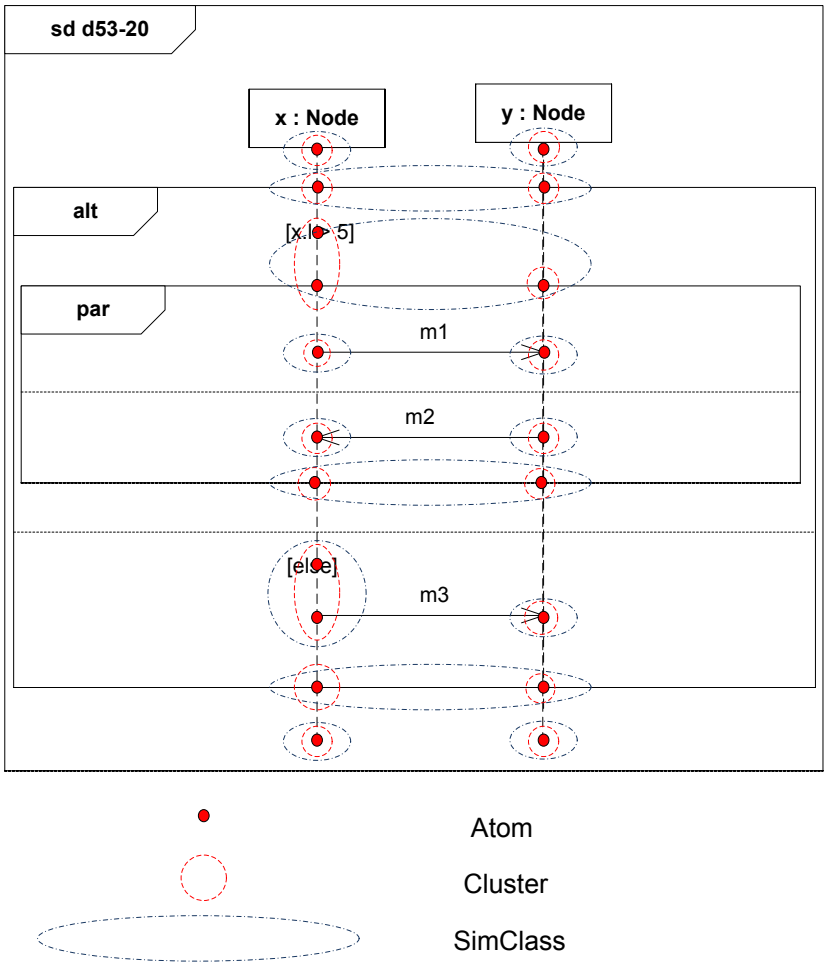


Figure 75: Atoms, clusters and Simclasses on a diagram

Two relations are defined between clusters on one Lifeline. Causality, denoted by ◻, defines a partial order between clusters. Conflict, denoted by #, defines which events cannot appear in the same trace, e.g. atoms from different operands of an *alt*.

Definition 6. Local causality: let *cl1*, *cl2* be two clusters on lifeline *l* with their location in the form: *location(cl1) = p1.i.p2* and *location(cl2) = p1.j.p3*, where *p1*, *p2*, *p3* can be the empty string.

$$cl1 \prec cl2 \text{ iff } j > i$$

Definition 7. Local conflict: let *cl1*, *cl2* be two clusters on lifeline *l* with their location in the form: *location(cl1) = p1.alt(i).p2* and *location(cl2) = p1.alt(j).p3*

$$cl1 \# cl2 \text{ iff } i \neq j$$

Definition 8. The predecessors function calculates the immediate predecessor(s) of a cluster *cl* on its lifeline *l*.

$$\text{predecessors}(cl) := \{cl' \in \text{Clusters}(l) \mid cl' \prec cl \wedge \neg cl'' \in \text{Clusters}(l) : cl' \prec cl'' \prec cl\}$$

For example, on Figure 74, the predecessor of the cluster with location 2.alt(1).0 is 1, while the predecessors of the cluster 2.alt(1).3 are 2.alt(1).2.par(1).0 and 2.alt(1).2.par(2).0.

For handling the causality between clusters on different Lifelines, the message sending and receiving events have to be mapped. To achieve this, every message is assigned a unique symbolic

id in the form *$i*, where *i* is an integer. Let *ID* be the set of identifiers generated that way, and let *MessageSends(sd) and MessageReceives(sd) be the sets of all message sending and* receiving atoms *of the diagram sd.*

Definition 9. The *messageID: MessageSends(sd)* $\cup$ *MessageReceives(sd)* $\rightarrow$ *ID* function returns, for each sending or receiving atom, the id of the corresponding message.

The *predecessors* function can be extended to *SimClass*es to contain also the causality relations implied by the connection between message sending and receiving. Let *SimClasses(sd)* represent the set of all *SimClass*es of diagram *sd*.

Definition 10. The immediate predecessors of a *SimClass scl* in the sequence diagram *sd* are given by the *prerequisite* function:

$$Prerequisite(scl) := \{scl' \in SimClasses(sd) \mid \exists cl \in scl, \exists cl' \in scl' : cl' \in predecessors(cl) \vee (\exists a \in cl \cap MessageReceives(sd), \exists a' \in cl' \cap MessageSends(sd) : messageID(a) = messageID(a'))\}$$

The conflict relation is extended to *SimClass*es using the conflict function.

Definition 11. The *conflict(scl)* : *SimClass* $\rightarrow$ $\wp$(*SimClass*) function returns the *SimClass*es which have clusters that are in different operands of an *alt* fragment than the operand in which the clusters of *scl* are.

We then have $scl_2 \in conflict(scl_1)$ if and only if the two SimClasses respectively contain a cluster cl_1 and a cluster cl_2 such that:

- the location of cl_1 on its lifeline l_1 has a *form* $prefix_1.k_1.alt(i).suffix_1$ *(i.e.,* cl_1 *is in an alt operand)*
- the location of cl_2 on its lifeline l_2 has a *form* $prefix_2.k_2.alt(j).suffix_2$ with j≠i *(i.e.,* cl_1 *is in an alt operand having a different number)*
- the *alt* coincide, that is, the following clusters belong to the same *SimClass*:
  - $cl'_1 \in Clusters(l_1)$ having location $prefix_1.k_1\text{-}1$
  - $cl'_2 \in Clusters(l_2)$ having location $prefix_2.k_2\text{-}1$

Note that the local conflict relation (#) corresponds to a special case of the global conflict, when $l_1=l_2$.

3.2.5.2 Unwinding algorithm

The aim of the unwinding algorithm is to build a symbolic automaton that characterizes traces as valid or invalid according to the requirement scenario. Inspired from [Klo], the principle is to gradually unwind the *SimClass*es of the diagram, until all of them have been processed.

The symbolic automaton is a tuple $(\Sigma, Q, q_O, F_T, F_S, \rightarrow, Var, Def)$ where:

- Σ is a set of transition labels with possibly symbolic variables in *Var*.
- Q is the set of states,
- q_O is the initial state,
- $F_T \subseteq Q$ and $F_S \subseteq Q$ are two disjoint subsets of accept states. They are used to distinguish trivial satisfaction of the requirements (the trigger before the Assert did not match) and stringent satisfaction (the content of the both the Assert and the trigger did match).

- $\rightarrow \subseteq Q \times \Sigma \times Q$ is the set of transitions.
- *Var* is the set of variables extracted from the TERMOS scenario. It includes all variables appearing in the spatial view (symbolic labels of vertices), in the event view (message parameters, free variables in OCL expression of guards and state invariants), and all symbolic message IDs *$i* produced by the preprocessing of the diagram.
- $Def \subseteq Q \times \wp(Variables)$ gives the subset of variables that are defined for each state. If $(q, \{v_1, v_2\})$ belongs to *Def*, then variables v_1, v_2 have a value in q, and all other variables are undefined in q.

The unwinding algorithm is based on the notion of phase, defined as a tuple *(Ready, History, Cut, Variables)* where:

- *History* is the set of *SimClass*es which have already been unwound,
- *Ready* is the set of *SimClass*es which are currently enabled to be unwound,
- *Cut* is a tuple $(cl_1, ..., cl_n)$ where each cl_j is a cluster from lifeline j. The current cut is intended to represent the borderline between already unwound elements and those that are currently enabled.
- *Variables* is the set of variables which are currently valuated.

The computed phases will correspond to automaton states. Like Klose, we assume that there is a function *STATE(ph: Phase)* assigning a unique state name for a phase. Note that if the same phase is encountered several times, the function is able to return the name already assigned at the previous steps of the unwinding algorithm.

The initial phase considered by the algorithm is $(History_0, Ready_0, Cut_0, Variables_0)$ defined as follows:

- $History_0 = \{\{\{\perp_1\}\}, \{\{\perp_2\}\}, \ldots, \{\{\perp_n\}\}\}$
- $Ready_0 = \{ scl \in Simclasses\ (sd) \mid Prerequisite\ (scl) \subseteq History_0\}$
- $Cut_0 = (\{\perp_1\}, \{\perp_2\}, \ldots, \{\perp_n\})$
- $Variables_0$ = set of variables appearing in the initial spatial configuration of the scenario (this includes the symbolic IDs of the nodes participating to the scenario).

That is, at the initial phase, only the lifeline heads have been unwound. We also assume that we start analysis in a state where the system is in the initial spatial configuration. This will be ensured by connecting the graph matching tool to the verification program (see Section 3.2.5.4). The graph matching tool also allows us to get concrete values for the IDs of nodes and all other variables defined by the initial configuration, which are then marked as valuated.

$STATE(Phase_0)$ is added to the set Q of the automaton states and to the F_T subset (Figure 76).

From the initial phase, the algorithm proceeds by computing successor phases (see Figure 77}. Given a phase ph = $(History_i, Ready_i, Cut_i, Variables_i)$, the *STEP* function returns the next phase obtained by firing a ready SimClass *scl* in a sequence diagram *sd*.

STEP (ph, scl) returns $ph' = (History_{i+1}, Ready_{i+1}, Cut_{i+1}, Variables_{i+1})$ defined as follows:

- $History_{i+1} = History_i \cup \{scl\} \cup conflict\ (scl)$, that is, both the fired *scl* and its conflicting SimClasses are considered unwound,
- $Ready_{i+1} = \{ scl' \in Simclasses\ (sd) \setminus \{\{\{T_1\}\}, \ldots\{\{T_n\}\}\} \mid$

Prerequisite (scl') ⊆ History$_{i+1}$ ∧ scl' ∉ History$_{i+1}$ }

- *Cut$_{i+1}$ = {cl'$_1$, ... cl'$_n$}* is produced from *Cut$_i$ = {cl$_1$, ... cl$_n$}* by letting *cl'$_j$ = cl$_j$* if lifeline *j* is not concerned by any cluster of the unwound SimClass. Other elements *cl'$_k$* are replaced by the corresponding cluster of the unwound SimClass, for each involved lifeline *k*.
- *Variable$_{i+1}$* is the union of *Variables$_i$* and of the set of newly valuated variables. Note that there are newly valuated variables only if *ph* contains communication events or configuration change events.

The new phase may, or not, correspond to an accept state. In the algorithm, this is governed by the *currentMode* variable. While the trigger is being matched, current mode is *AcceptTrivial* and the produced states are put in F_T. When the entering of an Assert box is unwound, current mode switches to *Reject*. It switches to *AcceptStringent* when the Assert is exited, and the successor is put in the set F_S. Transition labels and self-loops remain to be explained.

Roughly speaking, transition labels are obtained by conjoining the individual labels obtained from the atoms (of the clusters) of the unwound *SimClass* (The details are in Figure 78). If there are newly valuated variables, then the transition label also contains an explicit update action. For example, let us assume that we are currently unwinding a guard *"x>3"* and a send event *"(!m(x), n1, $4)"*. Let us also assume that the values for *x* and *n1* are currently defined, but not the symbolic message id *$4* assigned by the preliminary analysis. Then, the corresponding transition will be labelled: *"x>3 ∧ (!m(x), n1, $4) [update ($4)]"* which can be interpreted as follows: if *x>3* holds with the current valuation, and the next event of the trace can match *(!m(x), n1, $4)* with an appropriate assignment of *$4*, then the transition can be taken. Taking the transition consumes the event of the trace, and the current valuation is updated with the concrete message id of this event.

Entering and exiting boxes is simply represented by a *true* transition. Note that there could be further optimization to remove the unnecessary states.

Self loops must be added as soon as at least one of the exiting transitions contains a trace event, be it a communication event or a configuration change event. For example, if the next event of the trace does not match *(!m(x), n1, $4)*, are we allowed to consume this event and remain in the same state? Conversely if it matches, do we still have the choice to remain in the same state? The answer to the latter question is negative, hence the self-loop is labelled *¬(!m(x), n1, $4)*. The answer to the former question is generally positive according to our interpretation (see Section 3.2.2.1), but can be negative if the event is in the scope of a consider box. Let us also remind that our interpretation of *consider {m}* is (from Section 3.2.2.2): the sending of *m* is forbidden for all lifelines, but it is allowed to receive a message *m*, if its sending was not forbidden. Accordingly, the self-loop is concerned by sending events only. Finally, note that unexpected configurations change events are always forbidden, hence the self-loop label always contains *"¬CHANGE(-)"*.

```
// Initialization
Phases := {Phase0}
Q := {STATE(Phase0)} // set of states
q0 := STATE(Phase0)  // initial state
FT := {STATE(Phase0)}
FS := Ø
currentMode := AcceptTrivial
SelfLoopLabel := "¬CHANGE(-)" // unexpected config changes should never happen
// Unwinding loop: See Figure 77
```

Figure 76: Initialization of the unwinding algorithm

```
// Unwinding loop
While (Phases ≠ Ø)
   Extract ph = (History_i, Ready_i, Cut_i, Variables_i) from Phases
   If (Ready_i ≠ Ø)
      SelfLoop := false
      AddedSelfLabel = ""
      For all sc ∈ Ready_i
         successor := STEP(ph, sc)
         // compute the label of the triggered transition
         UpdatedVariables := Ø
         Label := ""
         For all cl ∈ sc
            For all a ∈ cl
               Switch a
                  entering of an assert box: See Figure 78
                     // Note: changes the CurrentMode to reject
                  exiting an assert box: See Figure 78
                     // Note: changes the CurrentMode to AcceptStringent
                  entering of a consider box: See Figure 78
                    //Note: changes SelfLoopLabel by forbidding considered send events
                  exiting a consider box: See Figure 78
                    //Note: changes SelfLoopLabel by discarding the forbidden events
                  entering or exiting a par or alt box: See Figure 78
                  Guard or state invariant: See Figure 78
                  Change in Configuration: See Figure 78
                     // Note: change UpdatedVariables to account for new variables in Ci
                     // Also, a selfloop is needed. SelfLoopLabel already contains
                     // a negated change event
                  Send or receive event: See Figure 78
                      // Note: change UpdatedVariables to account for new variables in
                      // the event
                      // Also, a selfloop is needed. If SelfLoopLabel does not already
                      // contain a negated form of the event (due to a consider),
                      // AddedSelfLabel is changed.
               End Switch a
            End For // all atoms of the cluster processed
         End For // all clusters of the unwound Simclass processed
         // Update the transition set
         If (UpdatedVariables ≠ Ø)
            Build a label ll of the form [list of updated variables]
            Append ll to Label
         Endif
         → := → ∪ {STATE(ph), Label, STATE(successor)}
         // Put successor in automaton states and in Phases
         Q := Q ∪ {STATE(successor)}
         If (CurrentMode = AcceptTrivial)
            F_T := F_T ∪ {STATE(successor)}
         Else if (CurrentMode = AcceptStringent)
            F_S := F_S ∪ {STATE(successor)}
         End if
         Put successor in Phases
      End for // All ready SimClasses processed
      // Add a self-loop if needed
      If (SelfLoop = true)
         If (AddedSelfLabel is not empty)
            Build label ll conjoining AddedSelfLabel and SelfLoopLabel
         Else ll = SelfLoopLabel
         Endif
         → := → ∪ {STATE(ph), ll, STATE(ph)}
      Endif
   Endif
End While
```

Figure 77: Unwinding loop

```
entering of an assert box:
   If (Label =””) then
      // first atom processed
      currentMode = reject
      Label = “true”
    Endif // Else nothing to do
exiting an assert box:
    If (Label =””) then
       // first atom processed
       currentMode = AcceptStringent
       Label = “true”
    Endif // Else nothing to do
entering of a consider box:
   If (Label =””) then
      // first atom processed
      for all considered message name m
         For all symbolic node id li in the current valuation
            build label ll of the form: ¬(!m(-), li, -)
            SelfLoopLabel := SelfLoopLabel conjoined with ll
         End For
      End For
      Label = “true”
      Endif // Else nothing to do
exiting a consider box:
   If (Label =””) then
      // first atom processed
      SelfLoopLabel := “¬CHANGE(-)”
      Label = “true”
   Endif // if not the first atom, nothing to do
entering or exiting a par or alt box:
   If (Label =””) then
      // first atom processed, or other atoms yielded a non empty Label
      Label = “true”
   Endif // if non empty label, no need to conjoin with true
Guard or state invariant
   Make a label ll with the predicate
   If (ll does not already appear in Label)
      Label := Label conjoined with ll
   Endif
Change in Configuration
If (Label =””) then
      // first atom processed
      UpdatedVariables := {variables in new config Ci} \ Variables_i
      SelfLoop := true
      Make a label ll of the form CHANGE(Ci)
      Label = ll
   Endif // if not the first atom, nothing to do
Send or receive event
   UpdatedVariables := {variables in message parameters or message id} \ Variables_i
   SelfLoop := true
   Make a label ll for the event
   Label := Label conjoined with ll
   If (the atom is receive event, or the atom is a send event that does not appear
                                  under the form ¬(!m(-), li, -) in SelfLoopLabel)
      AddedSelfLabel := AddedSelfLabel conjoined with ¬ll
   Endif // Else the event is already forbidden by an embodying consider
```

Figure 78: Processing of atoms

The use of this algorithm is illustrated by the scenario of the distributed black box application shown in Figure 72.

In Figure 79, we can see the automaton constructed from the sequence diagram. The states are the nodes q_i of the graph with the set of valuated variables (e.g. {n1,n2,n3,$1} for state q_1). In the figure, double circle nodes are representing trivial accept states, single circle nodes the reject states and triple circle nodes the stringent accept states.

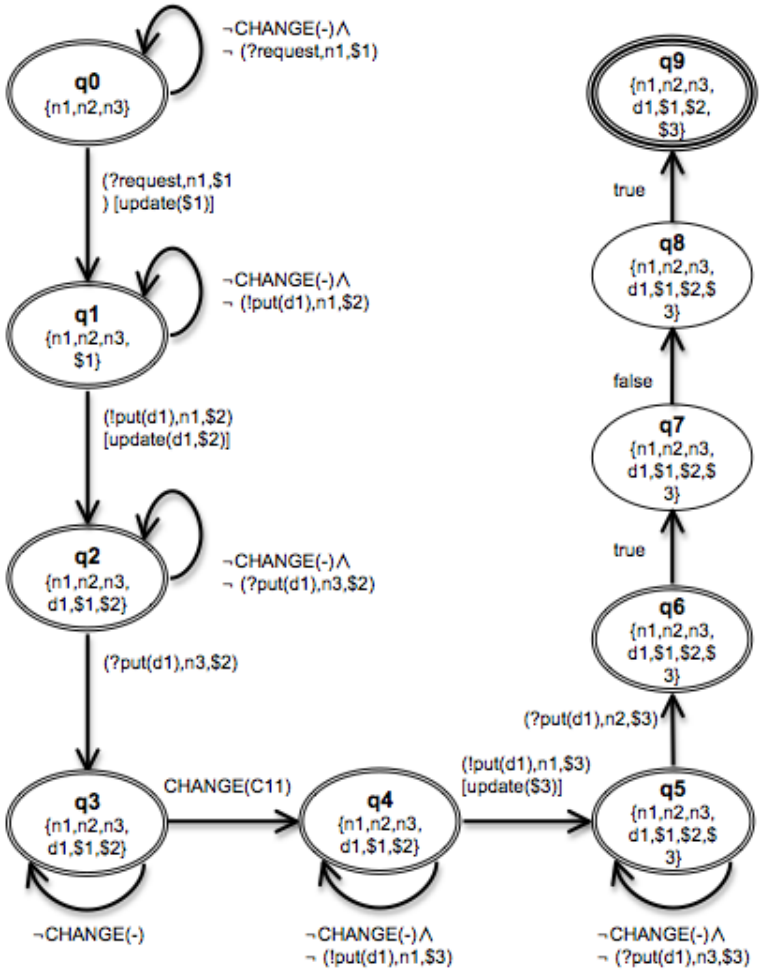


Figure 79: Automaton of DBB scenario

3.2.5.3 Well-formed diagrams

Well-formedness is not a purely syntactic issue. Some checks depend on the semantics. At the end of Section 3.2.3.4, we mentioned two checks:

- Check, whether messages depicted on the diagram can be sent and received in the current spatial configuration.
- Check, whether predicates refer only to message parameters received so far and to configuration labels from the current or past configurations.

The second check can only be performed on the automaton. It suffices to verify that, for each state q, the predicates appearing on the outgoing transitions do not refer to free variables that are undefined in q. The implementation is straightforward since, by construction, we know which

variables are defined for which state. The check could be integrated into the unwinding algorithm, i.e. when the transition labels for guard and state invariant atoms are computed.

The first check does not require consideration of the automaton. It can actually be performed as soon as the preprocessing step, when the orderings are computed. Using the position, it is straightforward to determine the current spatial configuration for a communication atom. Moreover, the message ID allows us to identify the sender of receiving events. It suffices then to verify that for each lifeline *i*:

- Each communication atom of *i* occurs in a configuration where *i* exists, i.e., there is a vertex with symbolic ID *i* in the configuration graph.
- If the atom is a receive event and lifeline *i'* of the diagram was the sender, then there is a path connecting *i* and *i'* in the current configuration graph such that all edge labels in the path have constant values.

Finally, there is the issue of whether the diagram may be ambiguous, that is, whether traces can be categorized as both valid and invalid. The automaton may exhibit states for which the outgoing transitions are not exclusive. For example, consider a state where two *SimClass*es are ready to be unwound, containing respectively a local invariant on lifeline *i* and a send event on lifeline *j*. There is a non-deterministic choice between evaluating the local invariant now, or delaying evaluation after the send event has been unwound.

This specific case should not be a problem, owing to the constraints put on predicate variables. If the corresponding well-definedness check passes, then the exact time for evaluating the predicate does not matter: whether it is evaluated now or later, the result will be the same. Note also that the *alt* operator should not be a problem either, because we impose a deterministic if-then-else form.

However, problems with the *par* operator are still possible. Assume that the diagram defines two parallel send events (!m(x1), n1, $1) and (!m(x2), n1, $2) that are ready at the same time. The verdict assigned to a trace:

(!m(1), “140.93.130.95”, 101) . (!m(2), “140.93.130.96”, 102) . …

may depend on which trace event is taken to match which scenario event, yielding either (x1:=1, x2:=2), or (x1:=2, x2:=1). We then propose the following check:

- For each state of the automaton with multiple outgoing transitions, verify that there is no pair of transitions having unifiable communication events.

Note that this check may generate false alarms. Back to the previous example, there will be a false alarm if variables *x1* and *x2* are both defined for the state, and the send events are in the scope of an *alt* operand guarded by $x1 \neq x2$. Hence, the check merely generates warnings and it is the responsibility of the user to determine whether a warning reveals a real problem.

3.2.5.4 Combining the spatial and event view

The analysis of the event view of a TERMOS scenario produces a symbolic automaton with variables that depend on the spatial configuration. The automaton must be instantiated in the framework of the concrete configurations that occurs during system execution. Checking whether a system trace satisfies or violates the scenario is done under the following conditions:

- Analysis is started in a state where the system is in a concrete configuration that matches the initial configuration of the scenario.
- The concrete values for the configuration variables, including the concrete IDs of nodes participating to the scenario, are known.

- The trace includes configuration change events.

Such conditions are fulfilled by using the GraphSeq tool. GraphSeq will be presented in the next section. It returns a set of matches for the desired sequence of spatial configurations where a match includes: (i) a valuation for all configuration variables, (ii) the temporal window for each individual configuration. This allows us to verify the trace against the scenario requirement as described in Figure 80.

```
For all matches returned by GraphSeq
   Extract the subtrace t of communication events occurring during the match
   Insert configuration change events according to the start date of each
                                                        individual configuration
   while (t is not empty and its first event occurs in the initial configuration)
      check whether t is accepted by the automaton
      let t = e.t'
      t:=t'
   End while
End for
```

Figure 80: Test oracle check against the scenario

3.3 GraphSeq: a graph matching tool

In our testing framework, scenarios descriptions involve graph constructs that describe the successive spatial configurations. The formal treatment of scenarios has thus to include graph matching algorithms. This section presents GraphSeq, the tool we have developed to search for the matches of spatial configurations in an execution trace.

GraphSeq uses graph homomorphism building as a core facility, as explained in Section 3.3.1. This allows us to determine whether one graph G1 (coming from a scenario) is matched by a subgraph of G2 (coming from a trace). Based on this facility, the tool reasons on sequences of graphs (i.e., sequences of spatial configuration). Some high-level principles of the sequential reasoning are provided in Section 3.3.2. After that, we gradually introduce the algorithms implemented in GraphSeq, starting from the case with a fixed number of nodes in scenarios (Section 3.3.3), and then adding consideration for nodes that are dynamically created and destroyed (Section 3.3.4). The validation of the tool is presented in Section 3.3.5, using both randomly generated graphs and outputs from a mobility simulator.

3.3.1 Graph homomorphism building as a core facility

Let L_V and L_E denote sets of labels for vertices and edges, and let $G = (V, E, \mu, \nu)$ denote a graph structure, where:

- V is the set of vertices,
- $E \subseteq V \times V$ is the set of edges,
- $\lambda : V \rightarrow L_V$ is a function assigning labels to the vertices,
- $\mu : E \rightarrow L_E$ is a function assigning labels to the edges.

A graph homomorphism is a mapping between two graphs that respects their structure. It can be mathematically defined as follows.

Definition. Let $G_1 = (V_1, E_1, \lambda_1, \mu_1)$ and $G_2 = (V_2, E_2, \lambda_2, \mu_2)$ be two graphs. A function $f : V_1 \rightarrow V_2$ is a graph homomorphism from G_1 to G_2 if and only if:

- It is injective,
- $\lambda_1(v_1) = \lambda_2(f(v_1))$ for all $v_1 \in V_1$,
- For any edge $e_1 = (v_{1s}, v_{1e}) \in E_1$, there exists an edge $e_2 = (f(v_{1s}), f(v_{1e}))$ such that $\mu_1(e_1) = \mu_2(e_2)$.

The definition captures the idea of G_1 being matched by a subgraph of G_2. In our work, G_1 is expected to come from a scenario description, and will be called the *pattern graph*. G_2 is extracted from a simulation trace and will be called the *concrete configuration graph*.

In practice, the basic definitions of graph structure and graph homomorphism need to be slightly extended to fulfil our needs. First, it may be convenient to assign tuple of labels to vertices and edges in order to allow a richer representation of nodes and relations between nodes. For example, assume that an application involves both mobile and infrastructure nodes. A node could be characterized by a 2-tuple <id, type>, where *id* would be a value uniquely identifying the physical node, and *type* would be an element of {Mobile, Infrastructure} that differentiates the mobile and infrastructure nodes. Second, we need to allow label variables in the pattern graph. In a scenario description, a node may be assigned labels <n1, Mobile> and it should be possible to detect a matching by a physical node <"140.93.130.95", Mobile> with substitution n1:= "140.93.130.95". As can be seen in this example, introducing variables means that the graph homomorphism building needs to exhibit a valuation that consistently unifies the labels.

The problem of graph homomorphism building has been extensively studied in the literature. It is thus possible for us to use an existing tool as the basis for the comparison of scenario descriptions and concrete traces. One of the existing tools has been developed by colleagues at LAAS-CNRS [Gue] in the framework of research on dynamically reconfigurable architectures. The tool searches for the set of all homomorphisms *(f,Val)* from a pattern graph G_1 to a concrete configuration graph G_2, where *f* is a mapping and *Val* is a valuation. In the definition of graph structures, the tool offers the following features (that were integrated in the definition of TERMOS, see Section 3.2.3.1):

- Vertices may be assigned at most 3 labels, yielding a 3-tuple of type STRING × INT× INT.
- Edges have at most one label of type INT.
- Label variables are supported for vertices only. Wildcards are supported for edges. We made a slight extension so as to have wildcards for vertices as well[7].

The complexity of the search is polynomial in the number of vertices of G_2, but exponential in the number of vertices of G_1 (which is not surprising, since the search problem is known to be NP-complete).

GraphSeq uses this tool as a core facility to search for matches in the case of *sequences* of graphs.

3.3.2 Reasoning on sequences of graphs

GraphSeq takes as input two sequences of graphs:

- A sequence $P_0, \ldots P_{m-1}$ of *m* pattern graphs,
- A sequence $C_0, \ldots C_{n-1}$ of *n* concrete configuration graphs.

It computes the set of all matches, where a match has the following data structure:

[7] Variables and wildcards are treated differently by GraphSeq. When trying to match sequences of graphs, variables are assigned a unique value, while wildcards are allowed to vary arbitrarily.

```
Structure Match
     Valuation val
     int index[0..m]
End Structure
```

The valuation *val* assigns a concrete value to every label variable appearing in the sequence of pattern graphs. Note that all pattern vertices have at least one label variable: the one corresponding to the STRING type. This label is intended to serve as an ID that uniquely identifies the corresponding system node, so that this node can be traced from one graph to the other. In the patterns, nodes have a symbolic ID, hence the label variable.

The index table gives the temporal window for each pattern. It is defined as follows:

- index[0] is the start date of the matching for P_0. That is, the matching starts at $C_{index[0]}$.
- For i>0, index[i] is the end date of the matching for P_{i-1}. That is, the matching ends at $C_{index[i]}$.

These notions are best illustrated by an example. Figure 81 shows a sequence of two patterns, as well as the beginning of a sequence of concrete configurations.

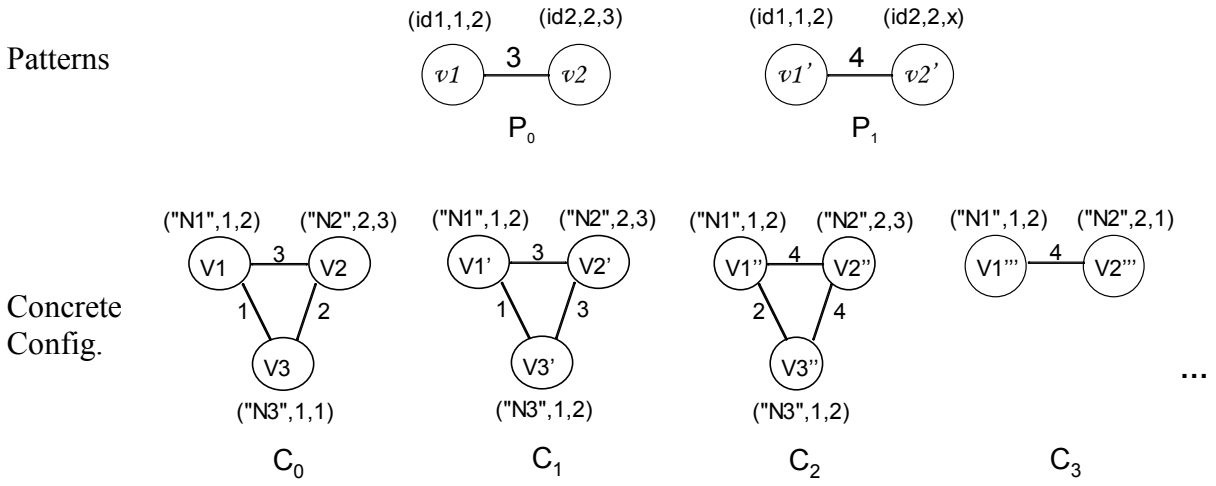


Figure 81: Example of graphs with a match starting at date 0 and ending at 2

One of the computed matches is:

val: { (id1,"N1"), (id2, "N2"), (x, 3) }

index:	0	1	2

The construction of this match uses homomorphism building as follows. First, P_0 and C_0 are compared. The tool finds a homomorphism (f_0, val_0) with:

f_0: *v1* → V1 val_0 = { (id1,"N1"), (id2, "N2") }
v2 → V2

The mapping f_0 involves vertex references *v1*, *v2*, V1, V2 that are specific to the encoding of the graphs. Here, the interesting information is the valuation provided to build the homomorphism: from now on, we will try to go on matching with concrete node N1 playing the role of abstract node id1, and N2 playing the role of id2.

We must now determine the end date for the matching of P_0. Does it persist at C_1? To check it, we must retain the valuation choices made at the previous step. For this, we use a basic utility offered by the LAAS tool: VALUATE_VERTICES(G,Val) takes as inputs a graph and a valuation, and rewrites all vertices according to the valuation. We create a pattern P_0' = VALUATE_VERTICES(P_0, val_0) and compare P_0' and C_1. The tool finds one homomorphism, meaning that the searched configuration persists in C_1. It is however no longer in C_2, hence index[1] = 1.

We go on with the next pattern of the sequence. Again, we need to retain the previous valuation choices. We thus search for a homomorphism from VALUATE_VERTICES(P_1, val_0) to C_2. The tool finds one with valuation val_1 = { (x, 3) }. We then merge the valuations found so far, yielding val = { (id1,"N1"), (id2, "N2"), (x, 3) }. The end date for matching is 2, because there is no homomorphism from VALUATE_VERTICES(P_1, val) to C_3: the last integer label of vertex V2''' does not have the expected value 3.

From what precedes, it is obvious that the main issue is to retain consistent valuation choices across the sequence. It may become tricky when nodes dynamically appear and disappear in the patterns. Figure 82 shows an example with node creation. For the sake of simplicity, the vertices of the graphs are labelled by their ID, and we omit all other labels. Assume that we are currently building a match that starts at *i* with valuation { (n1, "1") }. Pattern P_1 introduces a new node, with ID n2, that was not present in P_0. The problems are then the following:

- P_0 is matched by the concrete configurations until step *i+3* (and possibly later). Still, a transition to P_1 may be detected at intermediate steps *i+2* (appearance of concrete node "3"), *i+3* (appearance of concrete node "4") or later. These alternative choices must be taken into account to build the set of all possible matches.
- If transition to P_1 is searched at step *i+2*, then care must be taken not to retain concrete node "2" to play the role of n2. This is so because this concrete node already existed at step *i*, while it is required not to exist in configurations matching P_0.

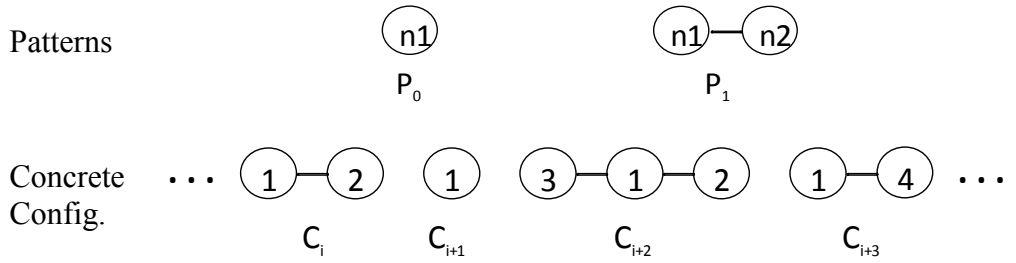


Figure 82: Example of problem with node creation

Such concerns complicate the search for matches. Hence, we first present the GraphSeq algorithm in the simpler case where patterns involve a fixed set of nodes. We then explain how the search is modified to account for nodes that appear and disappear.

3.3.3 Algorithm with a fixed set of nodes in patterns

GraphSeq uses the existing LAAS tool for building graph homomorphisms [Gue]. This tool is implemented in C++. It provides us with convenient definitions of data types:

- *Homomorphism* (and *ListOfHomomorphisms*),
- *Valuation*, which is used by Homomorphism,

- *Graph*, with all necessary facilities to encode constant labels, variable labels (for vertices) and wildcards (for both vertices and edges, with the slight extension we introduced).

The existing tool also provides us with the following functions:

- SEARCHHOMOMORPHISMS (G_1: Graph, G_2: Graph) that returns the list of all homomorphisms from G_1 to G_2.
- VALUATEVERTICES (G: Graph, V: Valuation) that returns a copy of G with the vertex labels rewritten according to the valuation V.
- MERGEVALUATIONS (V_1: Valuation, V_2: Valuation) that returns valuation $V_1 \cup V_2$ if its input valuations are compatible. If they are not compatible, i.e., they assign different values to a variable, the function returns NULL.

GraphSeq uses them to implement the search for matches. In order to gradually build matches, it uses an intermediate data structure called *PartialMatch*:

```
Structure PartialMatch
     Valuation val
     int index[0..m]
     int depth
End Structure
```

PartialMatch is thus like the Match structure presented in the previous section, but with an additional *depth* field. The depth value gives the number of patterns that have been successfully matched so far. For example, a depth of *i* indicates that patterns P_0, … P_{i-1} have been matched, but Patterns P_i, …, P_{m-1} are still to be processed. The values index[j] are meaningful only for $j \leq i$, since the end dates of unmatched patterns are not determined yet. In our implementation, they are assigned a spurious value -1. Back to the example of Figure 81, the first partial match found by GraphSeq is:

val: { (id1,"N1"), (id2, "N2") }

index:

0	1	-1

depth: 1

A partial match of depth *i* is extended by the processing of the next pattern P_i, yielding a partial match of depth *i+1*. For example, a one step extension to the previous partial match is:

val: { (id1,"N1"), (id2, "N2"), (x, 3) }

index:

0	1	2

depth: 2

When a partial match has been extended up to depth *m*, then a (complete) match has been found.

GraphSeq uses depth-first search (DFS) to extend partial matches. That is, if a partial match has several possible extensions, the tool will explore as far as possible along each branch before backtracking. The DFS control structure is shown in Figure 83. It uses a LIFO stack *L* to store the partial matches to be processed. Note that the outmost *for()* loop imposes that the matching of P_0

starts no later than date *n-m*. This is so because if it starts later, the remaining number of concrete configurations will be lower than the number of patterns to be matched.

```
Let L be an empty stack of PartialMatch elements
For (i=0; i≤n-m; i++)
   build all partial matches of depth 1 with start date index[0] = i,
   push each of them in L
   While L is not empty
      Let pm = pop (L)
      If (pm.depth < m)
         Build all one step extensions of pm,
         push each of them in L
      Else   // found
         Write pm.val and pm.index in output file
      Endif
   End While
End For
```

Figure 83: DFS control structure of GraphSeq

Given a candidate start date *i*, the search is initialized by looking at all possible partial matches for P_0, hence yielding a set of partial matches of depth 1. The corresponding algorithm is described in Figure 84.a, with auxiliary functions in Figure 84.b. Note that the valuation of an homomorphism yields a partial match only if:

- *i* is really a start date, and
- the end date is not too late.

Also, note that the checks of the dates use a valuated version of P_0.

```
// build all partial matches of depth 1
// with start date i,
// push each of them in L
Let H = SEARCHHOMOMORPHISMS (P0, Ci)
While H is not empty
   Extract h from H
   Let P' = VALUATEVERTICES (P0, h.val)
   // Is i a start date for P0?
   Let start_OK = true
   If (i > 0)
      Let H' = SEARCHHOMOMORPHISMS (P', Ci-1)
      If (H' is not empty)
         // previous config did match
         // --> not a start date
         start_OK = false
      Endif
   Endif
   // Now, check the end date
   If (start_OK = true)
      Let end = COMPUTEENDDATE(P', i)
      If (end ≤ n-m)
         // end date is not too late
         let pm = CREATEPARTIALMATCHD1 (h.val,
                                        i, end)
         push pm in L
      Endif
   Endif
End While
```

(a) Core algorithm for building the matches

```
int COMPUTEENDDATE (Graph G, int start)
   Let i = 1
   Repeat
      Let H = SEARCHHOMOMORPHISMS (G, Cstart+i)
      If (H is not empty) then
         i = i+1
      Endif
   until (H is empty or start+i = n)
   return (start+i-1)
End COMPUTEENDDATE

PartialMatch CREATEPARTIALMATCHD1 (Valuation v,
                                   int start, int end)
   Let pm be a Partial match
   pm.val = v
   pm.index[0] = start
   pm.index[1] = end
   For (i=2; i≤m; i++)
      pm.index[i] = -1
   End For
   pm.depth = 1
   return (pm)
End CREATEPARTIALMATCHD1
```

(b) Auxiliary functions

Figure 84: Partial matches of depth 1 starting at date i

The extension of a partial match of depth *d*, to produce partial matches of depth *d+1*, is described in Figure 85. Here, the start date is fixed: it comes just after the end of matching of the previous pattern. For each candidate continuation, the end date must be computed and checked. The check uses a valuated version of the pattern, after the valuations have been successfully merged.

```
// Build all one step extensions of pm,
// push each of them in L
// Here, pm.depth < m

Let start = 1+pm.index[pm.depth]
Let P' = VALUATEVERTICES (P_depth, pm.val)
H = SEARCHHOMOMORPHISMS (P', C_start)

While H is not empty
   Extract h from H
   Let v = MERGEVALUATIONS (pm.val, h.val)
   If (v!=NULL) // compatible valuations
      Let P'' = VALUATEVERTICES (P_depth, v)
      Let end = COMPUTEENDDATE(P'', start)
      If (end ≤ n-m+pm.depth)
         // end date is not too late
         let pm = CREATEEXTENDEDMATCH (pm, end,
                                       v)
         push pm in L
      Endif
   Endif
End While
```

(a) Core algorithm for extending the match

```
PartialMatch CREATEEXTENDEDMATCH (
  PartialMatch father, int end, Valuation v)
   Let pm be a PartialMatch
   pm.val = v
   For (i=0; i≤father.depth; i++)
      pm.index[i] = father.index[i]
   End For
   pm.index[father.depth+1] = end
   For (i= father.depth+2; i≤m; i++)
      pm.index[i] = -1
   End For
   pm.depth = 1+ father.depth
   return (pm)
End CREATEEXTENDEDMATCH
```

(b) Auxiliary function

Figure 85: Extending a partial match pm

3.3.4 Accounting for nodes that appear and disappear

Let us now account for sequences of patterns that involve varying sets of nodes. The DFS structure of the algorithm does not change, but the following parts are impacted (see Figure 86, with modifications indicated in bold):

- A preprocessing step is performed before the search is entered. Pattern graphs are analyzed to identify nodes that appear or disappear.
- The information extracted from the pre-processing step has an impact on both the computation of partial matches of depth 1, and their gradual extension up to complete matches.
- A final check is added before retaining a candidate complete match.

```
Preprocessing of pattern data
Let L be an empty stack of PartialMatch elements
For (i=0; i≤n-m; i++)
   build all partial matches of depth 1 with start date i,
   push each of them in L
   While L is not empty
      Let pm = pop (L)
      If (pm.depth < m)
         Build all one step extensions of pm,
         push each of them in L
      Else   // found
         If (FINALCHECK(pm))
            Write pm.val and pm.index in output file
         Endif
      Endif
   End While
End For
```

Figure 86: Impact on the DFS control structure of GraphSeq

The preprocessing step extracts, for each pattern P_j, the set N_j of symbolic node IDs that label the vertices. Then, it computes the following information to be stored in a global data structure:

- For $1 \leq j \leq m-1$, NewNodes(j) is the set of node IDs that are present in P_j, and did not occur in $P_0, .., P_{j-1}$.
- For $1 \leq j \leq m-1$, ForbiddenNodes(j) is the set of node IDs that occur at least once in $P_0, .., P_{j-1}$, but that are not present in P_j.
- ForbiddenNodes(0) is the set of node IDs that occur at least once in $P_1, .., P_{m-1}$, but that are not present in P_0. The meaning is thus different from the one of ForbiddenNodes(j) where $j>0$.
- For $0 \leq j \leq m-2$, StopBefore(j) is a Boolean value indicating whether the transition from P_j to P_{j+1} can occur before P_j ceases to be matched. Such is the case when the appearance of a node (either new or appearing again), is the only trigger for the transition: there is no node disappearing, and no inconsistency between constant labels in P_j and P_{j+1}. An example was given in Figure 82, where transition from P_0 to P_1 was caused by a new node having symbolic ID n2.

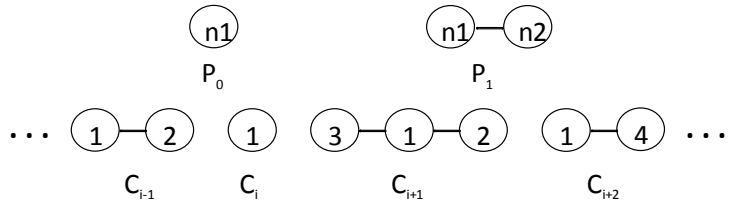
Figure 87 describes the corresponding computations. The initial extraction of the N_i, as well as the comparison of labels in the computation of StopBefore(i), are not detailed here because they depend on the precise encoding of the graphs in the existing tool.

The computed information is used to implement additional checks during the search for matches, as shown from Figure 88 to Figure 90 where the new parts are indicated in bold characters. If no node appears or disappears in the patterns, then the algorithms are equivalent to the ones shown in the previous section. Note that for a fixed set of nodes in patterns, NewNodes(j) and ForbiddenNodes(j) are empty, and StopBefore(j) is false for all j.

```
// Auxiliary computation AllNodes to be used for NewNodes and ForbiddenNodes
Let AllNodes(0) = N0
For (j=1; j≤m-1; j++)
   Let AllNodes(j) = Nj ∪ AllNodes(j-1)
End For

// New nodes
NewNodes(0) = ∅
For (j=1; j≤m-1; j++)
 NewNodes(j)= Nj \ AllNodes(j-1)
End For

// Forbidden Nodes
For (j=1; j≤m-1; j++)
   ForbiddenNodes (j)= AllNodes(j-1) \ Nj
End For
ForbiddenNodes (0) = AllNodes(m-1) \ N0

// Stop before?
For (j=0; j≤m-2; j++)
   If (Nj+1 ⊆ Nj)
      // No node expected to appear
      StopBefore(j) = false
   Else If (Nj ∩ ForbiddenNodes(j+1) ≠ ∅)
      // At least one node expected to disappear
      StopBefore(j) = false
   Else
      if there exists vertices v1 from Pj and v2 from Pj+1 such that
      v1 and v2 share the same symbolic IDs, and one of the vertex integer label
      is constant in both cases, but with different values
      // At least one vertex label is expected to change
      StopBefore(j) = false
   Else
      if there exists edges e1 from Pj and e2 from Pj+1 such that
      they connect pairs of vertices with the same symbolic IDs, and the edge label
      is constant in both cases, but with different values
      // At least one edge label is expected to change
      StopBefore(j) = false
   Else // transition from Pj to Pj+1 may occur before Pj ceases to be matched
      StopBefore(j) = true
   End if
End For
StopBefore(m-1) = false
```

Figure 87: Preprocessing of pattern data

The impact on the search for partial matches of depth 1 is shown in Figure 88. There are two major modifications, corresponding on additional checks of the start and end dates respectively. The check of the start date now depends on ForbiddenNodes(0). If this set is empty, decision is the same as in the previous section: we do not retain *i* as a candidate start date if C_{i-1} already matched P_0. But if the set is not empty, we always retain *i*. The reason for this decision can be explained by referring to the example in Figure 82, reproduced below for the sake of clarity:



In this example, *i* is retained because ForbiddenNodes(0) is not empty (it contains n2). Indeed, *i* is a start date for a complete match with valuation {(n1, 1), (n2, 2)}. Deciding whether i is really a start date cannot be done at depth 1 steps, because we would need the valuation for all symbolic IDs in ForbiddenNodes(0). Decision is taken by the final check we introduced (see the general control structure in Figure 86), that will be described later.

As regards the end date, the new part is when StopBefore(0) is true. In that case, we must accommodate several candidate end dates, ranging from the start date *i* to *min(end, n-m)*. A partial match is created in each case.

A last difference lies in the prototype of function COMPUTEENDDATE, which has now two additional parameters. At this step, the additional values ∅, NULL passed to the function induce the same behaviour as in the previous version. In the general case, the two parameters allow detection of end of matching when concrete configuration graphs contain any node in the list of forbidden ones. This facility is used when computing the 1-step extensions of a partial match.

```
// build all partial matches of depth 1
// with start date i,
// push each of them in L
Let H = SEARCHHOMOMORPHISMS (P0, Ci)
While H is not empty
   Extract h from H
   Let P' = VALUATEVERTICES (P0, h.val)
   // Is i a start date for P0?
   Let start_OK = true
   If (i > 0 && ForbiddenNodes (0)= ∅)
      Let H' = SEARCHHOMOMORPHISMS (P', Ci-1)
      If (H' is not empty)
         // previous config did match
         // --> not a start date
         start_OK = false
      Endif
   Endif
   // Now, check the end date
   If (start_OK = true)
      Let end = COMPUTEENDDATE(P', i, ∅,NULL)
      If (StopBefore(0))
      // can stop at an intermediate date
         For (j=i; j<=min(end,n-m); j++)
            let pm = CREATEPARTIALMATCHD1
                                 (h.val, i, j)
            push pm in L
         End for
      Else If (end ≤ n-m)
         // end date is not too late
         let pm = CREATEPARTIALMATCHD1 (h.val,
                                        i, end)
         push pm in L
      Endif
   Endif
End While
```

(a) Core algorithm for building the matches

```
ListOfHomomorphisms NEWSEARCHHOMOMORPHISMS
(Graph G1, Graph G2,SetOfStrings ForbidIds,
Valuation v)
   // None of the forbidden IDs in G2?
   For each f in ForbidIds
      Let concreteId be v(f)
      If (concreteId exists in G2)
         Return (empty list)
      End if
   End For
   Return (SEARCHHOMOMORPHISMS (G1, G2))
End NEWSEARCHHOMOMORPHISMS


int COMPUTEENDDATE (Graph G, int start,
       SetOfStrings ForbidIds, Valuation v)
   Let i = 1
   Repeat
      Let H = NEWSEARCHHOMOMORPHISMS (G, Cstart+I,
                                   ForbidIds,v)
      If (H is not empty) then
         i = i+1
      Endif
   until (H is empty or start+i = n)
   return (start+i-1)
End COMPUTEENDDATE


PartialMatch CREATEPARTIALMATCHD1 ()
   Same as previously
   See Figure 84.b
End CREATEPARTIALMATCHD1
```

(b) Auxiliary functions

Figure 88: Impact on building partial matches of depth 1 that start at date i

The new version of the extension of partial matches is shown in Figure 89. Note that the search for homomorphisms now takes the ForbiddenNodes sets into account (calls to NEWSEARCHHOMOMORPHISMS and to COMPUTEENDDATE). Lest us recall that when $1 \leq j \leq m-1$, ForbiddenNodes(j) is the set of node IDs that occur at least once in P_0, .., P_{j-1}, but that are not present in P_j. These IDs have already received a concrete valuation in partial match pm. We thus know the identity of nodes that must not be present in concrete configurations matching P_j and use this information to reject candidate extensions.

A special check for new nodes has been introduced. Their symbolic IDs receive a concrete valuation at the current step (in h.val). Care must be taken that the valuation really corresponds to new nodes, that is, there must be a check that the nodes did never appear in $C_{pm.index[0]}$, …, $C_{pm.index[pm.depth]}$.

Finally, the processing of the end date is made according to the value of the StopBefore Boolean, as already explained for partial matches of depth 1.

```
// Build all one step extensions of pm,
// push each of them in L
// Here, pm.depth < m

Let start = 1+pm.index[pm.depth]
Let P' = VALUATEVERTICES (P_depth, pm.val)
H = NEWSEARCHHOMOMORPHISMS (P', C_start,
                 ForbiddenNodes(pm.depth), pm.val)

While H is not empty
   Extract h from H
   Let v = MERGEVALUATIONS (pm.val, h.val)
   If (v!=NULL && CHECKNEWNODES (pm.depth,
         pm.index[0], pm.index[pm.depth], h.val))
   // compatible valuations
   // and new nodes are really new
      Let P'' = VALUATEVERTICES (P_depth, v)
      Let end = COMPUTEENDDATE(P'', start,
                    ForbiddenNodes(pm.depth), v)
      If (StopBefore(pm.depth))
      // can stop at an intermediate date
         For (j=i; j<=min(end,n-m+pm.depth); j++)
            let pm = CREATEEXTENDEDMATCH (pm, j, v)
            push pm in L
         End for
      Else If (end ≤ n-m+pm.depth)
         // end date is not too late
         let pm = CREATEEXTENDEDMATCH (pm, end, v)
         push pm in L
      Endif
Endif
End While
```

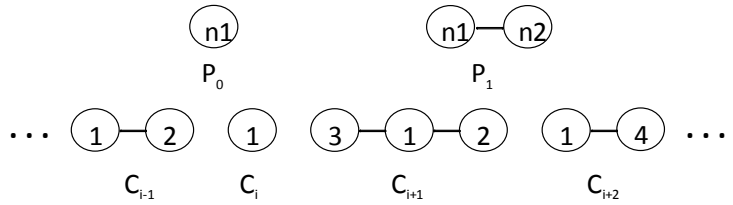
(a) Core algorithm for extending the match

```
Boolean CHECKNEWNODES (int i, int start,
                       int end, Valuation v)
   For each f in NewNodes(i)
      Let concreteId be v(f)
      // Is concreteId new?
      For (j=start, j≤end, j++)
        If (concreteId exists in C_j)
           Return (false)
        End if
      End for
   End For
   Return (true)
End CHECKNEWNODES


PartialMatch CREATEEXTENDEDMATCH ()
   Same as previously
   See Figure 85.b
End CREATEEXTENDEDMATCH
```

(b) Auxiliary functions

Figure 89: Impact on extending a partial match pm

It remains to explain the final check, shown in Figure 90. Remember that in the case where ForbiddenNodes(0) is not empty, we retained potential start dates *i* even if C_{i-1} already matched P_0. We must now decide whether or not i is a real start date. Decision will be negative if C_{i-1} matches P_0 and does not contain any of the forbidden nodes. Back to the example:



The check will accept *i* as a start date for a match with n2:="2", but not for n2:="3" or n2:="4".

```
Boolean FINALCHECK (PartialMatch pm)
   If (pm.index[0]=0 || ForbiddenNodes(0) = ∅)
      return (true)
   Else
      Let P' = VALUATEVERTICES (P0, pm.val)
      Let H= NEWSEARCHHOMOMORPHISMS (P', Cpm.index[0]-1, ForbiddenNodes(0), pm.val)
      If (H is empty)
         Return (true)
      Else return (false)
      Endif
   Endif
End FINALCHECK
```

Figure 90: Final check before outputting

3.3.5 Validation of GraphSeq

We performed a number of tests to validate GraphSeq. We first started with small examples that were manually produced (e.g., the example in Figure 82 was included as one test case), but quickly came to the conclusion that we would need an automated solution for both the generation of graphs and the analysis of results. We developed a tool that produces random sequences of graphs P_i and C_i such that, by construction, C_i contains at least one match. The GraphSeq results can then be automatically analyzed and a fail verdict is issued if the expected match is not found. Note that GraphSeq may find several matches, but the oracle check only concerns the one known to be there by construction.

The first version of the test tool produced test cases with a fixed set of nodes in patterns. We then extended the tool to accommodate nodes that appear and disappear. The random generation can be parameterized, and we produced about 900 test cases exhibiting various characteristics:

- Number of patterns from 1 to 5,
- Number of concrete configurations from 1 to 100,
- Number of nodes in patterns from 1 to 5,
- Number of nodes in concrete configurations from 1 to 25,
- For each individual P_i, duration of a matching from 1 to 20 steps in the concrete configurations.
- The transitions from P_i to P_{i+1} may involve a change in a node label, in an edge label, a node that appears, that disappears, or any combination (up to 5 changes).

The test tool proved very useful to debug GraphSeq, and to perform regression verification after changes in the C++ code.

We also experimented the connection of GraphSeq with a tool producing location-based data. We chose a mobility simulator developed at the University of South California, USA, as part of the IMPORTANT (Impact of Mobility Patterns On Routing proTocol in the mobile Ad hoc NeTworks)

framework [Bai]. The tool is freely available on the web[8]. It offers a rich set of parameterized mobility models, including Reference Point Group, Freeway and Manhattan mobility models. The generated traces are compatible with the ns-2 simulator [NS].

The connection of GraphSeq to such a mobility simulator requires the development of an interfacing component (see Figure 91) that abstracts the raw simulation data into a sequence of configuration graphs. In our experiments, the raw data are the position of nodes at each simulation step. Then, the abstraction consists in assigning edge labels according to the distance of nodes.

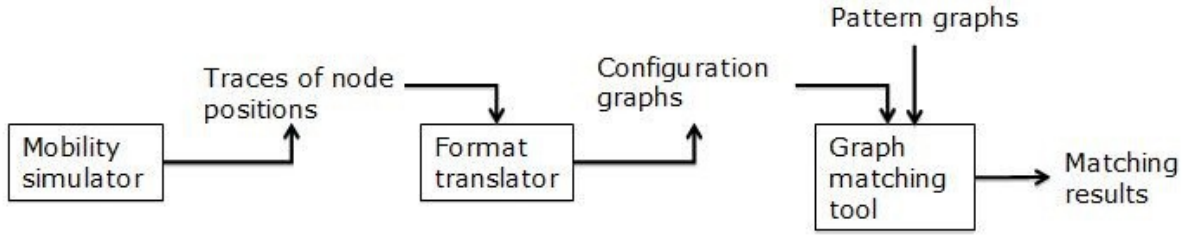


Figure 91: Connecting GraphSeq to a mobility simulator

We made trials with various mobility models and different parameterizations of the models. We describe here an example of run using the freeway model.

The freeway model has the following characteristics:

- Each mobile node is restricted to its lane on the freeway.
- The velocity of mobile node is temporally dependent on its previous velocity.
- If two mobile nodes on the same freeway lane are within the Safety Distance, the velocity of the following node cannot exceed the velocity of the preceding node.

Table 4 provides the parameters used for the example run. The map was built using predefined fragments of freeways and lanes, available in the simulation environment.

Parameters of the mobility model	**Value**
Number of simulation steps	350
Number of nodes	15
Acceleration (Max Speed/10)	4 $m.s^{-2}$
Max Speed	144 km/h
Map: Number of freeways Number of lanes	 2 6
Safety Distance	40 m
Transmission Range	300 m

Table 4: Parameters used for the simulation run

Our format translator extracts 350 configuration graphs from the simulation trace. The concrete ID of mobile nodes are labelled “N0”, “N1”, “N2”,… The edges representing the spatial relations between two mobile nodes depend on their distance *d* and are defined as follows:

- $0<d<140$m: edge = 1
- $140\leq d\leq 300$m: edge = 2
- $d>300$m: nodes are disconnected (300 m is the transmission range).

[8] http://nile.usc.edu/important/

The pattern graphs are arbitrarily defined as in Figure 92.

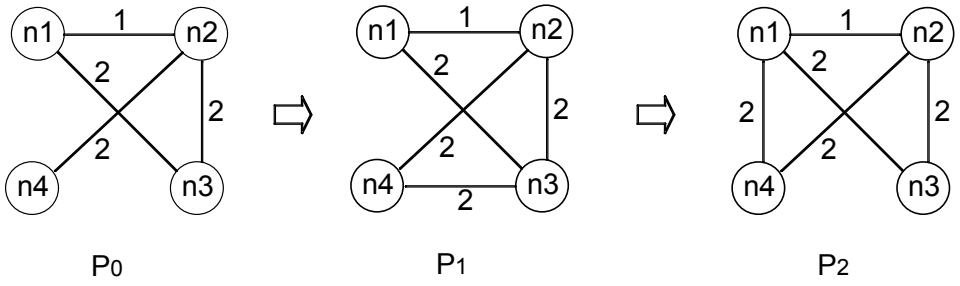


Figure 92: Pattern graphs of the example

GraphSeq is applied to search for the patterns in the sequence of concrete configurations, and returns the following result with four matches:

val: { (n1,"N14"), (n2, "N6"), (n3, "N0"), (n4, "N5") }

Index:	124	127	141	153

val: { (n1,"N13"), (n2, "N6"), (n3, "N0"), (n4, "N5") }

Index:	124	127	140	153

val: { (n1,"N13"), (n2, "N6"), (n3, "N3"), (n4, "N5") }

Index:	126	136	140	153

val: { (n1,"N14"), (n2, "N6"), (n3, "N3"), (n4, "N5") }

Index:	128	136	141	153

3.4 Conclusion of the testing contribution

The testing activities in HIDENETS focused on verifying whether applications running in mobile environments fulfil their high-level requirements. Based on the review of the state of the art and on a testing case study, we identified the main open research challenge to be the lack of an adequate formalism to capture system-level behaviour and spatial topology in a mobile setting. For this reason, the two main contributions of the work have been:

- the definition of a language to capture the mobility-related specificities in system-level interaction scenarios,
- the development of some automated treatment for matching test traces with the specified scenarios.

Scenarios may serve several roles to support testing activities, and our contribution focused on the definition of requirements to be tested. A new scenario language, called TERMOS, has been proposed. The language is based on UML, the same language used for specifying the applications in the design framework in Section 2. In this way the models capturing the test requirements can be included next to the design models. They are complementary to the latter models in the sense that

they provide a system-level view, while the design adopts a node-centric perspective. The technical details of the TERMOS language definition can be summarized as follows:

- UML Sequence Diagrams have been analyzed in detail and have been tailored to the need of a deterministic, verifiable, test requirement language that accounts for mobile settings;
- An abstract and concrete syntax based on UML metamodelling have been proposed for TERMOS, as well as a formal semantics based on automata.

Example scenarios taken from HIDENETS use cases and the testing case study were given to demonstrate the language concepts.

The formal semantics of TERMOS makes it possible to consider the automated verification of test traces against requirements. The symbolic automata produced from the sequence diagrams must be instantiated with parameters that depend on the spatial configurations of the system nodes. A graph matching tool, called GraphSeq, has been designed and developed to extract these parameters from the traces. Raw location-based data are first abstracted by labelled graphs, and GraphSeq then reasons on sequences of such graphs. The technical results include:

- The development of algorithms for handling sequences of configurations, with possibly mobile nodes appearing and disappearing in the test traces. These algorithms have been implemented with the help of an existing facility for graph homomorphism building.
- The resulting tool, GraphSeq, has been validated using test data produced by random traces and by traces created with a mobility simulator.

The coupling of the TERMOS language and the GraphSeq tool should provide a useful way to express and validate requirements in mobile settings. GraphSeq could also be used in association with other language variants that would concentrate on other testing aspects, e.g. test purposes or test cases in mobile settings. While such language variants have not been investigated in HIDENETS, we believe that the experience gained during the design and development of TERMOS would greatly facilitate their definition.

4 Summary

The work to "the extension the state of the art software engineering methods and tools in order to cope with the specific requirements of highly dependable mobile system design" [HDoW] resulted in the fulfilment of the common target. It was carried out in two cooperating tasks: (1) "UML design patterns and workflow" developing a UML based design methodology for mobile applications and (2) "Testing methodology and framework" developing methodologies to support the testing of resilient mobile applications and services. The cooperation with the other work packages of the project was productive:

- From *WP1 (Use case scenarios and reference model)* we have obtained the identified use case scenarios and dependability requirements. With our application, middleware, domain concept and scenario models we have contributed to the reference model and fault model.
- From *WP2 (Resilience architecture and middleware)* and *WP3 (Resilient communication)* we have obtained the middleware (the middleware, oracle and communication services, and the middleware architecture). With our formalisation work we have contributed to the consistency and completeness analysis of the middleware. Our application development and testing frameworks provided basis for means and tools of the future industrial utilization of the middleware.
- For *WP4 (Quantitative Evaluation)* we have provided application and middleware models and their semi-automatic translation into quantitative models. The testing framework and the holistic evaluation approach provide complementary analysis methods.
- In *WP6 (Proof-of-concept experimental set up)* our application development approach, development supporting tools and design patterns are demonstrated in the ADTB.

In the last 2,5 years several researchers, doctorate and graduate students worked together at three project partners in three different European countries, spending about 65 person-months to gain new results and to publish them in project deliverables, scientific journals, book chapters, conference and workshop presentations, technical reports and diploma theses. The results inspire both further scientific research and industrial development.

Our original concept, basing both of our application development framework and testing framework on model driven methods, is proved by the results. We have defined

- an application development framework that enables the efficient use of the HIDENETS middleware for application developers without deeper knowledge in the implementation details of the single services and
- a testing framework that provides an adequate formalism to capture system-level behaviour and spatial topology in a mobile setting

This deliverable documents the main results of our work:

- elaborating an application development methodology that helps application designer in the understanding and effective utilization of the (dependability, mobility and communication related) domain knowledge that is manifested in the HIDENETS middleware,
- defining a UML profile incorporating the peculiarities of this environment and allowing a semi-formal formulation of user requirements and basic architectural solutions,

- formulating design patterns to support the direct reuse of the HIDENETS architecture and middleware solutions while application development,
- the definition of a deterministic, verifiable, test requirement language (TERMOS) based on the UML Sequence Diagrams to capture the mobility-related specificities in system-level interaction scenarios,
- the development of the algorithms and tooling of some automated treatment for matching test traces with the specified scenarios.

Our work justifies the wide applicability of the model driven concept in special environments and for working with special requirements, and that a common basic modelling notation can support collaboration even if the actually applied modelling language extensions are different.

Appendix A

UML Based Modelling and Metamodelling

As nowadays' systems are getting more-and-more complex new ways had to be found to keep the control over the creation and maintenance of them. As a result, different modelling languages have been created that provide abstract, domain specific views and thus simplify the overview of these tasks. One of these languages is the *Unified Modeling Language*.

The Unified Modelling Language (UML) [UMLsup][UMLinf] is a standard of the Object Management Group (OMG). UML is a visual language for specifying, constructing and documenting the artefacts of systems. It is a general-purpose modelling language that can be used with all major object and component methods, and that can be applied to all application domains and implementation platforms. During the last few years UML has emerged as the software industry's dominant modelling language. It is widely accepted among system designers, analysts and programmers. The UML specification is defined using a metamodelling approach that adapts formal specification techniques. While this approach lacks some of the rigor of a formal specification method, it offers the advantages of being more intuitive and pragmatic for most implementers and practitioners.

UML was designed as a general modelling language. However, instead of defining all the modelling concepts of the domains where UML could be used, the specification contains only some core elements and a standardized extension mechanism is given. These extensions include the *Constraint*, *Stereotype* and *TaggedValue* constructs. A constraint is an expression that restricts the structure or the behaviour of an element, usually written in the Object Constraint Language (OCL) [OCL]. A stereotype defines how an existing class may be extended, and enables the use of platform or domain specific terminology or notation in place of or in addition to the ones used for the extended class. A stereotype can be attached to a modelling element to further classify it and add additional properties to it. A tagged value is a property defined for a stereotype.

Metamodelling is the precise definition of a modelling language. A metamodel consists of the concepts and rules that can be used in a model for a specific problem. More precisely, for the definition of a modelling language the followings shall be given i) the abstract syntax defining the concepts of the given domain and their relations, ii) the concrete syntax defining the textual or graphical notations of the concepts, iii) well-formedness rules defining further constraints for the concepts, and iv) the formal semantics defining the dynamic behaviour of the models.

One of the first metamodelling frameworks is OMG's standard Meta-Object Facility (MOF) [MOF]. The MOF specification defines an abstract language and a framework for specifying, constructing, and managing technology neutral metamodels. A metamodel is in effect an abstract language for some kind of metadata. Examples include the metamodels for UML, CWM, SysML and the MOF itself.

The specification of MOF includes the following aspects:

- a formal definition of the MOF meta-metamodel; that is, the abstract language for specifying MOF metamodels,
- an XMI format for MOF metamodel interchange.

The XML Metadata Interchange (XMI) [XMI] specification defines technology mappings from MOF metamodels to XML DTDs (Document Type Definition) and XML documents. These

mappings can be used to define an interchange format for metadata conforming to a given MOF metamodel.

UML and MOF are normally viewed in the context of a conceptual layered metadata architecture. Although the metamodels for MOF and UML are designed to be architecturally aligned, sharing a common subset of core object modelling constructs, this does not bind the modeller to stick to UML as the modelling language. Just on the contrary, the whole metamodelling mechanism is useful to provide a common modelling framework where model instantiation can occur using different modelling languages.

The classical framework for metamodelling is based on an architecture with four metalayers. These layers are conventionally described as follows:

1. the information layer with the data that should be described;
2. the model layer with an abstract representation of the data in the information layer;
3. the metamodel layer with the descriptions that define the structure and semantics of metadata;
4. the meta-metamodel layer with the description of the structure and semantics of meta-metadata.

Figure 93 depicts this classical four layer framework illustrated with a HIDENETS, Platooning use case related example. The metamodel layer contains a metaclass (PhysicalNode) taken from the metamodel discussed below. The model layer presents a fragment of a software model building on the HIDENETS metamodel (introducing classes FirstVehicle and FollowingVehicle as possible members of a platoon) while the information layer presents an actual platoon (in the form of an object diagram) where the platoon consists of a first vehicle and two following vehicles.

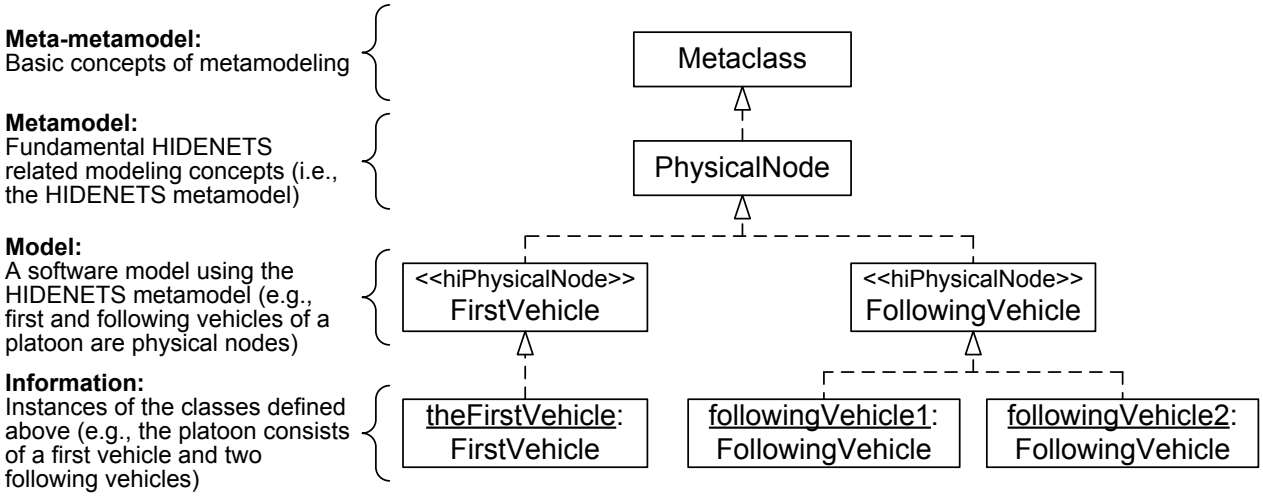


Figure 93: Illustration of the MOF 4-layer Framework

As the first adopted technologies specified using a metamodelling approach, the UML, MOF, and XMI provide the foundation for OMG's Model Driven Architecture (MDA). Future metamodel standards should reuse MOF and UML's core semantics and emulate their systematic approach to architecture alignment.

Specifications of the Service Availability Forum

The Service Availability™ Forum (SA Forum) [SAF] aims at providing standardized solutions for making services highly available. The *Application Interface Specification* (AIS) [AIS] of the Forum

defines the standard interfaces for accessing Highly Available (HA) middleware and infrastructure services that reside logically between applications and the operating system. All AIS services are designed to operate in a cluster environment where computation nodes are connected to each other and work together for a specific goal. The interfaces are defined in the form of sub-specifications, namely:

- **Availability Management Framework (AMF)**

  The Availability Management Framework is a general approach for high availability needs in environments which run redundant components. Its goal is to ensure application availability by detecting component failures and shifting service load from failed components to sane components.

- **Checkpoint Service (CKPT)**

  A *checkpoint* contains application-specific data partitioned in *sections*. It is a cluster-wide entity designated by a unique name. A checkpoint is made highly available by *replication*. The Checkpoint Service is responsible for the handling and the replication of checkpoints. The application component requests the creation, the update and the deletion of checkpoints which are replicated according to configured or application-defined rules.

- **Cluster Membership Service (CLM)**

  The Cluster Membership Service provides information about the current cluster configuration and the nodes that are members of the cluster. The cluster consists of a set of configured nodes.

- **Event Service (EVT)**

  The Event Service permits an M:N communication between event *publishers* and event *subscribers*. It supports the distribution of information (by the publishers) to a set of "interested" applications (the subscribers), that can select this information according to specified filter criteria. Communication takes place over *event channels*. Multiple publishers and subscribers can communicate over the same event channel.

- **Information Model Management Service (IMM)**

  The Information Model Management Service provides the information base of all objects handled by services attached to it. It is intended as a repository especially for the SA Forum services, but not restricted to them. It keeps information about various objects belonging to the attached services, e.g. the configuration and runtime attributes of services (called *service units* here), checkpoints, message queues, etc.

- **Lock Service (LCK)**

  The Lock Service provides cluster-wide *lock resources* and the ability to set or release locks on them. Locks are used to synchronize accesses from competing processes or nodes to shared resources.

- **Logging Service (LOG)**

  The Log Service provides interfaces through which applications can act as *loggers*. They can log events of different categories (alarms, notifications, system information, application information) into cluster-wide resources maintained by the Log Service, the *log streams*.

- **Message Service (MSG)**

The Message Service provides *queues* and *queue groups* for the M:1 resp. M:N communication between Message Service clients within a cluster.

- **Naming Service (NAM)**

  The Naming Service provides the storage and the retrieval of named objects.

- **Notification Service (NTF)**

  The Notification Service provides APIs to work with *notifications* in a *producer*, *subscriber* or *reader* role. A notification is a data structure that describes an important event (in its natural meaning, not in the one of the SA Forum Event Service) during the lifetime of an HA cluster.

- **Timer Service (TMR)**

  The Timer Service provides a mechanism by which client processes get notified when a *timer* expires. A timer is a logical object that is dynamically created and represents either absolute time or duration (i.e. an interval relative to a time reference point).

For more detailed description of these services see the corresponding section in [D2.1].

The SA Forum Information Model. The entities defined in the AIS specifications (e.g. service units, message queues, applications, etc.) are described semi-formally by the SA Forum Information Model (IM) in the form of UML classes.

From the perspective of the HIDENETS project the most relevant entities are the ones defined by the Availability Management Framework specification. In the following we introduce the metamodel of the AMF entities.

The AMF is the software entity that provides service availability by coordinating redundant resources within a cluster to deliver a system with no single point of failure. AMF defines two types of entities, the *physical* and the *logical entities*.

Physical entities. Every *physical entity* managed by the Availability Management Framework is a *resource*. These physical entities are either hardware equipment or software abstractions implemented by programs running on that hardware. In Figure 94 the hierarchy of physical entities is depicted. The **Resource** class represents the resource abstraction and it is also the base class for the specialized resource entities. These entities are the following:

- the **Physical Node**, which represents a computer with an operating system;
- the **Local Resource**, which represents a local resource from fault containment point of view, so that if the host physical node fails all of the hosted local resources become inoperable;
- all other resources are called **External Resource**s, and failures of external resources are independent of physical node failures.

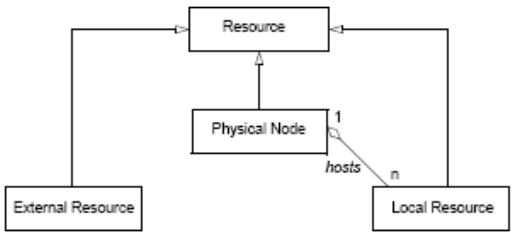


Figure 94 : Physical Entities of the Availability Management Framework

Logical entities are the abstract bricks of a high availability service. Each level of the service can be represented by a logical entity. The *component* is the smallest logical entity on which the Availability Management Framework performs any action, and the *application* represents the highest level of the service. The hierarchy of the AMF logical entities is depicted in Figure 95.

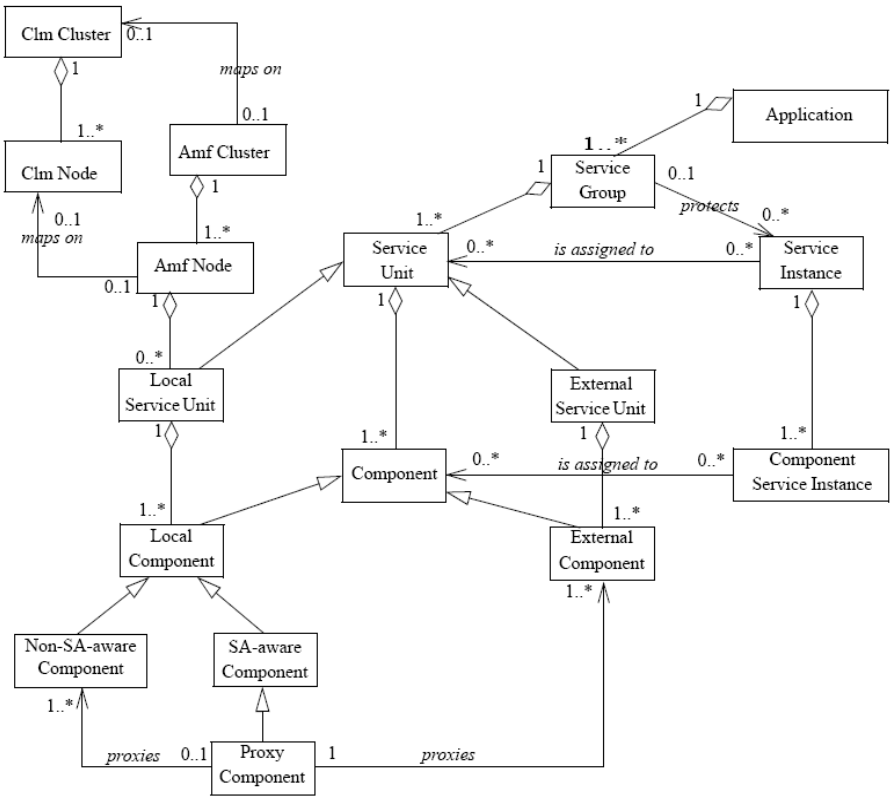


Figure 95 : Logical Entities of the Availability Management Framework

The **Component** represents a set of resources to the Availability Management Framework. The resources represented by the component encapsulate specific application functionality. This set can include hardware resources, software resources or a combination of the two. It is the smallest logical entity on which the Application Management Framework performs error detection, isolation, recovery, and repair. The scope of the component must be small enough so that its failure has as little impact as possible on the services provided by the cluster. Furthermore, the component should include all the important functions that cannot be separated.

The Component class is an aggregated notion that may refer to either a *Local* or an *External Component*. A **Local Component** represents a subset of the local resources contained within a single node while an **External Component** represents a set of resources that are external to the AMF cluster.

The *Local Component* class is further specialized into **SA-Aware** and **Non-SA-Aware Component** classes that refer to whether the given *Local Component* implements the interfaces that enable the AMF to monitor the health of the component or not. In case of *Non-SA-Aware Components* and *External Components* a **Proxy Component**, which is a special *SA-Aware Component*, has to be used that uses proprietary communication methods to forward the health check requests of the AMF to the designated components.

A **Service Unit** is a logical entity that aggregates a set of *Components* combining their individual functionalities to provide a higher level service. A *Service Unit* can contain any number of *Components* but a given *Component* can be configured in only one *Service Unit*. Using this model, the *Components* can be developed in isolation, and the developer might be unaware of which *Components* constitute a *Service Unit* since they are defined at deployment time only.

From the perspective of the Availability Management Framework the *Service Unit* is the unit of the redundancy so that it is the smallest logical entity that can be instantiated in a redundant manner.

*Service Unit*s are aggregated into **Service Groups**. The *Service Group* prescribes the manner in which the *Service Units* are instantiated in order to make the individual services that are provided by the *Service Units* highly available. This manner is called the redundancy model. The redundancy model of the *Service Group* can be for example the "2N", which means a simple failover behaviour, or the "N+M" model for N active, M standby behaviour.

As mentioned above, the **Application** represents the highest level of the service, which is provided by the cluster. It contains one or more *Service Groups* and combines the individual functionalities of the constituent *Service Groups* to provide the higher level service. An *Application* can contain any number of *Service Groups*, but a given *Service Group* can be configured in only one *Application*.

A **Component Service Instance** represents the workload that the Availability Management Framework can dynamically assign to a *Component*. High availability (HA) states are assigned to a *Component* on behalf of its *Component Service Instances*.

Each *Component Service Instance* has a set of attributes (name/value pair), which characterize the workload assigned to the *Component*. These attributes are not used by the Availability Management Framework, and are just passed to the *Components*. The Availability Management Framework supports the notion of *Component Service Instance Type*. All *Component Service Instances* of the same type share the same list of attribute names.

In the same way as *Components* are aggregated into *Service Units*, the Availability Management Framework supports the aggregation of *Component Service Instances* into a logical entity called a **Service Instance**. A *Service Instance* aggregates all *Component Service Instances* to be assigned to the individual *Components* of the *Service Unit* in order for the *Service Unit* to provide a particular service.

When a *Service Unit* is available to provide service, the Availability Management Framework can assign HA states to the *Service Unit* for one or more *Service Instances*. When a *Service Unit* becomes unavailable to provide service, the Availability Management Framework removes all *Service Instances* from the *Service Unit*. A *Service Unit* might be available to provide service but not have any assigned *Service Instance*.

The Availability Management Framework assigns a *Service Instance* to a *Service Unit* programmatically by assigning each individual *Component Service Instance* of the *Service Instance* to a specific *Component* within the *Service Unit*.

AMF Cluster and Node entities. An **AMF Node** is the logical representation of a *Physical Node* that has been administratively configured in the Availability Management Framework configuration. An *AMF Node* is also a logical entity whose various states are managed by the Availability Management Framework using designated administrative operations that are defined for such nodes.

The complete set of *AMF Node*s in the Availability Management Framework configuration defines the **AMF Cluster**. The *AMF Cluster* is one of the entities that are under the Availability

Management Framework control, and its various states are managed by the Availability Management Framework. There are Availability Management Framework administrative operations that are defined on the *AMF Cluster*.

The restart of an *AMF Node* will only stop and start entities under Availability Management Framework control, without any impact on the cluster membership. The restart of the *AMF Cluster* will restart all *AMF Nodes*.

Applications to be made highly available are supposed to be configured in the Availability Management Framework configuration. Each *Application* is configured to be hosted in one or more *AMF Nodes* within the *AMF Cluster*.

One or more *Service Units* can be assigned to an *AMF Node* to provide service.

Compliance to HIDENETS objectives: The Application Interface Specification standardizes the access interfaces to several commonly used services such as the checkpointing or messaging. These interfaces cover the required functionalities of a HIDENETS node in the infrastructure domain, therefore it was decided that any HIDENETS node in the infrastructure domain has to offer these standard interfaces. In this way our design support can utilize the advantages of the existing standard.

Industrial relevance: SA Forum specifications are widely accepted in the industry. There is a dynamically growing number of technology adapters and implementers.

Extensibility: The SA Forum interfaces are designed to be used as is and thus there is no support for customizing them.

Extension requirements: During the HIDENETS project the weakness in supporting mobility was identified. The project results were presented to the standardization body where the work on the extension started.

Tool support: Since the SA Forum specifications are emerging standards currently there is no wide range of tools that support the development for SA Forum based systems.

Openness: The SA Forum specifications are freely accessible and there are several open source implementations e.g. OpenAIS [OpenAIS], OpenSAF [OpenSAF].

Appendix B

The following figure depicts the abstract syntax of the test requirement language.