Compliance Verification of a Design Model with respect to its Specification Model in the Context of Software Defined Radios: a Model Transformation Approach.

by

Juan Pablo Zamora Zapata

B.C.S., M.B.A.

A Thesis submitted to the
Faculty of Graduate Studies and Research
in partial fulfillment of
the requirements for the degree of
Doctor of Philosophy

Ottawa-Carleton Institute for Computer Science

School of Computer Science
Carleton University
Ottawa, Ontario

January, 2005

Your file Votre référence
ISBN: 0-494-00809-1
Our file Notre référence
ISBN: 0-494-00809-1

Dedicated to my wonderful parents,
Carlos and Alma Leticia,
my beautiful wife,
Martha,
and our beloved daughters
Paola and Valeria.

Acknowledgments

First of all, I wish to record my most sincere gratitude to my thesis Co-Supervisors Professors Francis Bordeleau and Jean Pierre Corriveau. Extremely bright minds, they provided me with years of invaluable support for my research. They spent countless hours reviewing my work and gave me feedback to which I'm deeply grateful for. Both are wonderful human beings and they excel at what they do. To Dr. Bordeleau and Dr. Corriveau I say: You know how much I appreciate everything you have done for me... Thanks! I wish you, and your families, health and happiness in your lives.

I want to thank Professor Evangelos Kranakis for his support while I was on the process of joining the program at the School of Computer Science. He oppened our lives to a much better future. My family will always remember what he did for us.

My gratitude goes also to the members of my Ph.D. thesis committee: Professors Daniel Amyot, Dorina Petriu and Bran Selic. I thank them for reviewing my thesis and providing me with valuable feedback. I would also want to thank Dr. Robert France for his acceptance to be the External Examiner of the Ph.D. thesis committee. Acknowledging all of your busy schedules thanks again for taking the time to be at my thesis proposal, and later at my final dissertation.

I want also to thank the following friends and colleagues who provided me with technical assistance and friendship during my student days: Miguel Vargas for his invaluable support in helping me to join the Ph.D. program and for tutoring me throughout the degree; Gerardo Reynaga for his true friendship and for making himself always available to help (he was extremely competent in computer related technical issues as well as in home improvement projects); Javier Govea for you kind help in the earlier years while I was taking courses. Of course I cannot forget my other colleagues:

Loreto Bravo (is there any Internet tool that you are not aware of?); Monica Caniupan (I'm sure we both enhanced our Spanish vocabulary); Natalia Villanueva (you brought a piece of México with you); Tobin McClean and David Arnold (there will be a day in the near future without a 3004 project).

I would like to express my heartfelt gratitude to my parents for allowing my sister, brothers and me to pursue our dreams since our earliest days in life. It turned out that my sister Claudia, my brothers Carlos and Pedro, and myself shared the dream of the highest education. At the end of 2005, the net result will be four Bachelor degrees (Industrial Relations, Architecture, Computer Science and Literature), four Master degrees (three MBAs, one MFA), and two Ph.D.s (one in Computer Science and one in Hispanic Literature). For what we have achieved, we are proud to be your children.

Many thanks to México's Consejo Nacional de Ciencia y Tecnología (National Council for the Sciences and Technology) for the financial support that enabled me to study in Canada. I also thank the following organization for their financial help at some point during the degree: the School of Computer Science at Carleton University, the Communications Research Centre Canada, Mercury Computer Systems, Inc., the National Sciences and Engineering Research Council of Canada, and the Communications and Information Technology Ontario.

I'm also grateful to the School of Computer Science at Carleton University for providing me the nice environment in which I worked. In particular, I thank Maureen Sherman, Linda Pfeiffer, Nicole Enouy, Sandy Dare, Claire Ryan and Jane MacArthur.

In closing, I lack the words to thank God for giving me the health and strength required to complete this journey, but most importantly, for allowing me to meet (almost 10 years ago) the most wonderful woman on Earth. I married her two years later and she has since been at my side through thick and thin. We have formed a beautiful family and she is the foundation of it. I thank God, again, for putting her on my path. This Thesis would not have been possible without her unwavering support. Here goes to you and our little girls...

Abstract

International organizations such as the Object Management Group (OMG), the International Telecommunication Union (ITU) and the International Organization for Standardization (ISO) define standard specifications against which future products will be developed and ultimately verified. Such is the case of the *Specification for PIM and PSM for Software Radio Components* (SRC) being developed by the OMG's Software-Based Communication (SBC) Domain Task Force.

The OMG's Model Driven Architecture (MDA) provides a framework for model driven development. The notion of platform independence (i.e., PIM vs. PSM) and the notion of level of abstraction (i.e., specification vs. design) constitute two distinct facets of software modeling. The former is concerned with the dependency that a model has on specific technology platforms (e.g., CORBA, .NET, etc.), while the latter is concerned with the level of details and the completeness a model offers. Both specification and design models can be considered PIMs or PSMs, depending on the inclusion or exclusion of platform specific details such as middleware technology (CORBA, J2EE, .NET). Whereas current MDA research mainly focuses on the transition between PIMs and PSMs, we are interested in the more general problem of compliance between specification and design models. A solution to this problem requires considering two aspects of product development, namely: 1) realization of a standard specification model into a design model, and 2) verification of the compliance of a design model 'against' a standard specification model. It is this latter issue that we address in this thesis, developing a case study using the SRC specification.

In essence, our proposal splits compliance between a specification and a design into two separate steps. First, the original specification is to be transformed into a

semantically equivalent model (we call it *tSpec* for transformed specification) expressed in the language used to define the design. This transformation specifically addresses the semantic gap between the specification metamodel and the design metamodel. Second, a design model is to be obtained and its compliance to the *tSpec* verified. We are interested in structural and behavioral compliance. For each activity considered in our proposal, we discuss its automation and its consequences for compliance.

Table of Contents

List of Acronyms

Acronym	Description
CASE	Computer Aided Software Engineering
COTS	Commercial Off-The-Shelf
CWM	Common Warehouse Metamodel
CIM	Computation Independent Model
CPT	Correctness Preserving Transformation
DTF	Domain Task Force
ISO	International Organization For Standardization
ITU	International Telecommunication Union
JTRS	Joint Tactical Radio System
JPO	Joint Tactical Radio System Joint Program Office
Jtel	Joint Tactical Radio System Technology Laboratory
MOF	Meta Object Facility
MDA	Model Driven Architecture
MDD	Model Driven Development
MSRC	Modular Software-Programmable Radio Consortium
OCL	Object Constraint Language
OMG	Object Management Group
PIM	Platform Independent Model
PSM	Platform Specific Model
QVT	Query/View/Transformation
RFP	Request For Proposal
RM-ODP	Reference Model Of Object Distributed Processing
ROSE-RT	Rational Rose RealTime
RQA	Rational Quality Architect
SBC	Software-Based Communication
SCA	Software Communications Architecture
SDR	Software Defined Radio
SDRF	Software Defined Radio Forum
SRA	Software Radio Architecture
SRC	Software Radio Components
SWRadio	Software Radio
UML	Unified Modeling Language
XML	Extensible Markup Language

List of Figures.

List of Tables.

Chapter 1. Introduction

In this introductory chapter we present a high level overview of our thesis research. We start by describing our context and motivations. We then define the problem we are trying to solve followed by our proposed solution. We continue by presenting the method that we propose to solve our problem and how we are validating our approach. We then discuss the applicability and generality of our approach. We continue by briefly listing the contributions of our thesis and conclude this introductory chapter with an outline of the remaining chapters of this document.

1.1 Context and Motivation: Specification – Product Development – Compliance Verification

In this section, we focus on the activities pertaining to the use of a standard specification from which future software products will be developed and ultimately verified. This is a concrete context for research as it applies, in particular, to the use of the specification of a standard (e.g., for software radios) from which different vendors will develop distinct products. We postulate that such a development process involves a) so-called 'realization' activities that modify such a specification into products, and also b) verification activities that ensure that a resulting implementation does behave as defined in the standard specification.

Our research work is at the convergence of two core software engineering areas and an emerging communication technology that is leading the way for the next generation of communication devices. As Figure 1 illustrates, Software Defined Radios is that emerging technology and the two software-engineering areas are Model Driven Development and Software Testing. Our thesis research is focused on the small black area where the circles intersect (magnified on the right side of the figure). We call it

'Compliance Verification of a Software Defined Radio Design model with respect to a Software Defined Radio Specification model'.

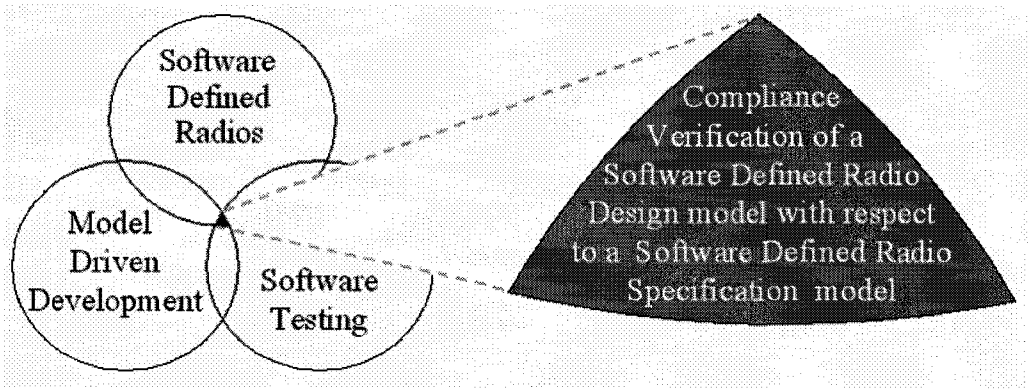


Figure 1. Thesis research focus of interest.

Transformation and verification activities are not new in software engineering. What is new to the field is Model Driven Development (MDD), which constitutes the latest proposal for automatically transforming models into other models (and ultimately into implementations). Automated transformations have been attempted before with mitigated success [25, 30, 26], especially when Computer Aided Software Engineering (CASE) tools gathered a lot of attention at the end of the 80's. The existence of the UML framework constitutes a key difference between CASE and MDD. MDD is promising mostly because it rests on industry-wide standards and mature technologies [90] that automatically generate programs (from their correspondent models), which is a key premise of MDD. We briefly elaborate on this topic also in Chapter 2.

Within MDD, the Object Management Group (OMG) through its Model Driven Architecture (MDA) initiative is proposing a conceptual framework to define standards in support of MDD [90]. In the particular context of MDA, specifications may be written in the form of Computation Independent Models (CIMs), Platform Independent Models (PIMs) and Platform Specific Models (PSMs), that is, models that are independent or not of a computational technology or platform on which the future implementations are to execute. This conceptualization promotes the use of specification level models not only as guidelines to follow, but also as input for automatic generators that will refine

specifications by implementing model-to-model mappings. If we consider a transformation as the application of one or several mappings, then after a series of model-to-model transformations, implementations are typically automatically generated using model-to-code mappings. In this context, we refer to a model from which an implementation (i.e., code) is to be derived as a *Design* model.

1.2 General Problem Definition

In the context of MDA, a key question to ask is how to assess if a design model is compliant with a standard specification model? We contend that an answer to this question likely requires considering the two aspects of product development mentioned in section 1.1, namely: 1) realization of a standard specification model into a design model, and 2) verification of the compliance of the design model 'against' the standard specification model. Figure 2 gives a high level view of such a process, which provides the context for our research (there are no semantics defined for the arrows used in the figure).

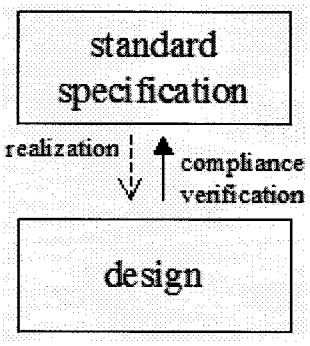


Figure 2. Specification to Design Realization/Compliance Process

We observe that the issue of compliance, which is at the center of this dissertation, pervades the development of actual industrial applications. We will emphasize this point by presenting an extensive case study (in Chapter 4) that illustrates the problem, and our proposed solution, in the context of Software Defined Radios (SDR) (see section 2.3).

It is crucial to understand that we restrict ourselves here to proposing an approach that refines the compliance activities of Figure 2. In particular, we view the standard specification and design models as inputs to these activities that we want to study. Our specific goal is to suggest a method to go about verifying the compliance of one (or many) design(s) against a unique standard specification. We do want to avoid any qualitative judgment of the inputs per se, and instead aim at describing the activities that, we claim, help in deciding whether or not a design is compliant with a standard specification. Consequently, we want to stay away from a) the issue of evaluating any aspect of 'quality' of the standard specification and of the design models, as well as b) the considerable literature available on this vast problem. Furthermore, because we postulate that both the standard specification and the design models are inputs[1] to our approach, we will also avoid discussing purely (one-way) transformational approaches that go (via one or more semantic preserving transformations) from a specification to its corresponding design model. Round-trip engineering, although related to our work, is also considered out of the scope of this thesis since that approach is also based on an input model to produce or update an output model. That is, we want to address the issue of compliance between two input models, which is a quite different problem from proving that a design/specification model was correctly derived from a specification/design through a series of transformations. Finally, we do not pursue a formal verification approach because our specification and design models are not defined with the level of formality required for such purposes. We do not close the door, though, and acknowledge that formal verification may be considered in the future to complement our proposed approach.

In the rest of the document we refer to a standard specification model simply as specification model.

[1] Supplied by third parties

1.3 Overview of Proposed Solution

We postulate that a compliance process that attempts to directly link a specification to a design model, as outlined in section 1.2, may be quite difficult to realize in practice. This is due to the fact that, typically, there is a semantic 'gap' between specification and design models. Trying to build direct compliance links between elements of a given specification model and elements of a given design model will likely be highly challenging, to say the least. The problem, in our opinion, is that direct links between elements of a specification and elements of a design are hard to determine because they bypass the origin of the semantic gap. We observe that different aspects contribute towards creating such a semantic gap, and we propose to treat them separately. For example, we propose different mechanisms to address the fact that generally, a) specification and design models may use different representation languages (metamodels), b) that the specification is 'more abstract' than the design, c) that the design implements more than one specification, and d) that the design may only implement a subset of a specification. We further illustrate the different aspects that can contribute to the semantic gap between a specification model and a design model in section 2.1.

Figure 3 presents our proposed solution to this problem. In essence, our proposal splits compliance between a specification and a design into two separate steps. First, the original specification is to be transformed into a semantically equivalent[2] model expressed in the language used to express the design. This transformation specifically addresses the semantic gap between the specification (or source) metamodel and the design (or target) metamodel. We will discuss its automation, as well as its consequences for compliance. That is, we will explain how to go about deriving a so-called transformed specification (hereafter *tSpec*) from a specification (hereafter *Spec*), and then how to verify that the former is compliant with the latter.

Second, a design model is to be obtained and its compliance to the transformed specification verified. Based on our experience with current industrial practices, we

[2] We define semantic equivalence in Chapter 3.

repeat that we assume that the realization from a *Spec/tSpec* to a design (hereafter *Design*) is carried out by a designer in an *idiosyncratic* way and thus lies beyond the scope of our research. That is, a designer takes inspiration from the *Spec* and/or its equivalent *tSpec* to *manually* produce a *Design* (represented by the dotted arrows in Figure 3). The creative nature of this activity makes it considerably less amenable to automation. Our proposal addresses how to verify the compliance of this design with respect to a specification and its corresponding *tSpec*. More precisely, we will refine the compliance verification link from a *Design* to a *tSpec* into a series of activities to be overviewed in the next section.

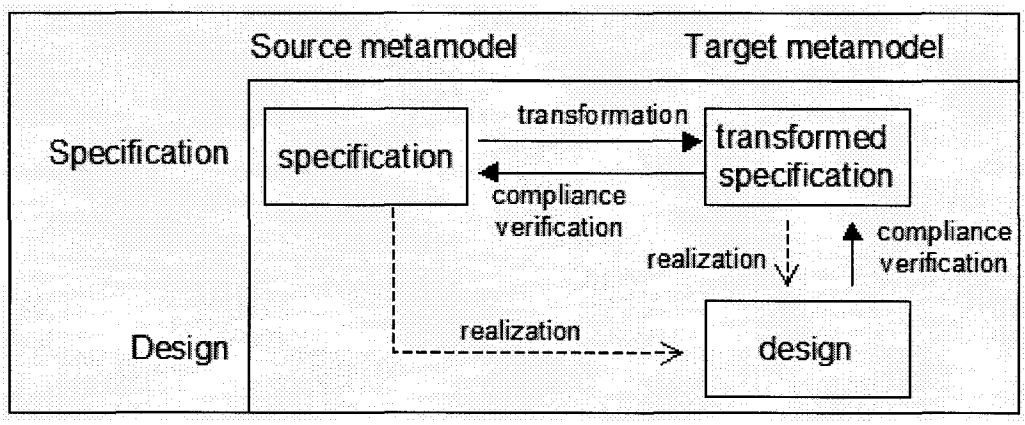


Figure 3. Two-step approach for Transformation/Compliance

We have questioned ourselves about the necessity of the intermediate *tSpec* model. While we believe that model compliance verification may be feasible without it, we have considered that separation of concerns and 'simplicity' aspects sufficiently motivate its use. Let us make an analogy with the building of executable code for several programs. The process first compiles each individual program and generates intermediate files that are later used in a linking process that eventually produces executable code. The scope of the compilation task is kept mostly to the internals of each program, while the linking process targets the resolution of external references to elements defined in other programs. Is it possible to build executable code in a single pass? We believe so. Is such a process more complex than the actual compilation-linking approach? Undoubtedly. Then again we reiterate that although we recognize that a single step process may be

feasible, we opted for a less complex process that also provides for a clearer distinction of activities.

Let us consider the different roles required for the realization, transformation and compliance activities illustrated in Figure 3. For *Spec* to *tSpec* transformation and compliance tasks, deep knowledge is needed about the *Spec Domain*, as well as the languages (metamodels) being used to define the *Spec* and *Design* models. This role is carried out by what we call *Spec Designers*. Realization from *tSpec* to *Design* (which again, is considered beyond the scope of this thesis) implies different design activities. To differentiate this role from the previous one, we refer to these designers as playing the role of *Spec Implementers*. Finally, *Spec Verifiers* carry out activities related with the verification of compliance of a Design with respect of a *tSpec* model. *Spec Verifiers* are commissioned (by the organization that defines the standard) to evaluate *Design* models as compliant or non-compliant with the standard specification. *Spec Designer* may (or may not) be part of the organization that defines the standard specification.

1.4 Our proposal and its validation

In this section, we first briefly overview the activities in each of the two steps we propose and then address how to validate our approach to the verification of the compliance of a *Design* against a *Spec* and its corresponding *tSpec*.

1.4.1 From Specification to Design, and back

Within the conceptual framework of Figure 3 we now sketch out the activities that we will focus on in this dissertation.

Step 1: *Spec* to *tSpec* transformation/compliance:

- *Source and target metamodel definition (or selection).* The first goal is to capture the type of elements and relationships used in the *Spec* and *Design* models in order to determine the source and target metamodels.

- *Derivation of mappings to bridge the gap between source and target metamodels.* The objective here is to obtain the definition of the semantic relationships that hold between elements of the source and target metamodels. These mappings are to be automated and the resulting tool must be capable of producing a target model from a source model.
- *Verification of the target model.* The goal of this activity is to answer the following questions: Have the mappings been correctly applied to the source model in order to create the target model? Are all the elements in the source model used as originators of elements in the target model?

Step 2: *tSpec* to *Design* realization/compliance:

- *Design model definition.* This task lies beyond the scope of this work. The *Design* models used in this dissertation only serve the purpose of addressing their compliance with a specification (see section 3.2). In other words, because they are only illustrative, we will not discuss how the *Design* models are obtained, nor any qualitative evaluation of them.
- *Structural compliance of a Design model with respect to a tSpec model.* The goal here is to ensure that the structural elements of a *Design* (and their properties) do 'correspond' to those of the relevant *tSpec*.
- *Behavioral compliance of a Design model with respect to the tSpec model.* In this thesis we limit the scope of behavior compliance verification to behavior expressed by means of UML sequence diagrams. We assume a *Spec* and its corresponding *tSpec* express behavior by means of UML Sequence diagrams (as is the case in the case studies). The goal of this activity is to verify that a *Design* model can execute a series of message interchanges defined in the *Spec* model.

1.4.2 Validation of the Approach

The transformation and compliance approach outlined in Figure 3 is, first and foremost, validated through an extensive case study, as is generally the case in software engineering theses. We emphasize that this case study focuses on an industrial domain, namely, software radios. In particular, we do reuse the initial specification provided for

this domain (as will be explained in Chapter 4). Beyond this case study per se, we have automated several facets of our proposal, which we believe strengthens the soundness of our work. More specifically, we offer tool support for the following tasks:

- The automatic transformation of a *Spec* into a *tSpec* through the application of *Spec Designer* (as opposed to hardwired) mappings.
- The verification of the correct application of such mappings. This activity checks that the outcome of the application of the mappings matches the expected results as defined by the *Spec Designer.*
- Checking that all elements from the *Spec* model have been taken into account as originators of elements of the *tSpec* model.
- Structural comparison between a *Design* and a *tSpec* model
- Verification (through a commercial CASE tool) that a *Design* can execute the behavior captured in the sequence diagrams of a specification.

1.5 Applicability and Generality of the Proposed Approach

We consider that standard specifications defined as models (using UML or any other modeling language with well-defined semantics) are suitable to use our approach for model compliance verification. Recalling from section 1.1, the OMG is leading the way in the definition of such standard specifications. Table 1 presents some OMG standard specifications that may benefit from the application of our approach when considering the verification of compliance of design models. The first column presents the name of the specification. The second column shows the status of the specification within the OMG process. Finally, the third column contains the OMG document number for reference purposes. The specifications are ordered by their status in the OMG standardization process with the more advanced ones (i.e., the ones closer to becoming OMG standards) at the top of the table.

Table 1. OMG standard specifications suitable to use our compliance verification approach.

Standard Specification	Status in the OMG process	OMG document
Data Distribution Service	Revised Final Adopted Specification	ptc/04-04-12 [66]
Software Radio Components	Final Adopted Specification	dtc/04-05-04 [73]
Super Distributed Object	Final Adopted Specification	dtc/03-09-01 [74]
Product Lifecycle Management	Draft Adopted Specification	dtc/04-05-05 [72]
Business Process Management Facility	Revised Draft RFP	bei/02-06-01 [63]
Business Process Runtime Interfaces	RFP	bei/02-06-08 [64]
Product Data Management	Draft RFP	mfg/02-06-02 [71]
A PIM Framework for Telecom OSS	Proposal to OMG	telecom/02-06-06 [70]

We tailored our approach based on the contents of the first four specifications (from Table 1) that hold the status of adopted specifications. In particular, for structural compliance, we took into account the elements and relationships used to define these four specifications. We expand on this discussion in Chapter 5, after we have explained the elements and relationships used throughout our case study.

For behavioral compliance, we considered the formalism used to represent the behavioral content of these specifications. Table 2 presents the usage of UML sequence and statechart diagrams in each of the first four specifications. The first column presents the OMG specification. The second column presents the number of sequence diagrams, followed in columns three and four by the average number of lifelines and messages per sequence diagram respectively. Columns five through seven present the number of statechart diagrams in the specification and the average number of states and transitions per statechart diagram in each specification. At the bottom of the table we present the average usage for each column considering the four standard specifications.

Table 2. Usage of Sequence and Statechart diagrams in four OMG specifications.

Standard Specification	Sequence Diagrams			Statecharts		
	#	Lifelines (average)	Messages (average)	#	States (average)	Transitions (average)
Data Distribution Service [66]	3	7	30	8	3	7
Software Radio Components [73]	20	4	5	4	2	3
Super Distributed Object [74]	7	3	7	0	0	0
Product Lifecycle Management [72]	5	3	8	0	0	0
Average	9	4	13	6[3]	3	5

[3] The average numbers for statechart diagrams consider only the two specifications that make use of them

From Table 2 we observe that while sequence diagrams are used in the four specifications, only two of them make use of statechart diagrams. Also we note that the average number of sequence diagrams per specification is greater that the average number of statechart diagrams. Finally, the average number of lifelines and messages used in sequence diagrams makes us believe that the complexities of such diagrams capture more meaningful and enforceable behavior than the statechart diagrams. Based on these observations, we decided on the use of sequence diagrams as our behavioral formalism for our model compliance verification approach. The latter does not prevent the use of statecharts within our approach as we further explain in section 5.1.

We consider behavioral compliance as the capability of a design model to execute a series of message interchanges as defined in the specification level sequence diagrams. We found that extensive research has been done with respect to comparing a sequence diagram that results from an actual execution of a model against a previously defined sequence diagram. We decided on the use of available off-the-shelf tools to carry of such comparison. Several commercial tools support such executability including Rational Rose RealTime © (ROSE-RT) [33] in combination with Rational Quality Architect (RQA) [33], Objecteering Tests for Java/EJB © [61], I-Logix Rhapsody © [31], and Telelogic Tau © [100]. Consequently, work on sequence diagram comparison was considered a different area of research and thus beyond the goals of this thesis.

1.6 Contributions

In our opinion, the following are the most significant contributions of our work:

- The proposed *Design* to *Spec* compliance verification approach (and its activities).
- The software radio transformation and compliance verification case study. In particular, the mappings to a ROSE-RT metamodel used in it.
- The principles behind our prototype tools, which were developed in the context of the case study (and others using the same metamodels).

In Chapter 5 we discuss the generality of our contributions and propose a series of steps to execute our approach using different standard specifications and/or different behavioral formalisms.

1.7 Outline for the rest of this dissertation

In Chapter 2, we present the background material on which this thesis rests. In Chapter 3, we explore at length our two-step transformation/compliance approach (outlined in Figure 3). The chapter describes in detail each of the activities of our proposed method, and also discusses their automation. Also within Chapter 3, we present the rationale (and examples) of our mapping definitions to transform specification models expressed in UML 1.5 into semantically equivalent models expressed in ROSE-RT. Chapter 4 illustrates our proposal by applying it to the industrial domain of software radios. In Chapter 5 we discuss the generalization of our approach into other specifications. In Chapter 6 we offer a summary of the thesis, as well as some conclusions and ideas for future work. Finally Appendix A illustrates results of compilation of a SWRadio *Design* model. The appendix is preceded by a bibliography.

Chapter 2. Background

In this chapter we present the background material on which this thesis rests. We start with a brief review of the nature and origin of the gap that may exist between specification and design models. We continue with a survey on international standards that are used in or have influenced our research. We conclude this background chapter with an introduction to Software Defined Radio (SDR), which provides a context for our case study.

2.1 Specification vs. Design

International organizations like the Object Management Group (OMG), the International Telecommunication Union (ITU) and the International Organization for Standardization (ISO) define standard specifications against which future products will be developed and ultimately verified. Standard specifications and their implementations are usually at different abstraction levels. Standard specifications are more abstract than their more concrete implementations. Somewhere in the middle, design models capture concrete solutions for the abstractions of the specification. Developing products that conform to a standard specification requires transforming abstract specifications into design models that drive the development of concrete implementations.

An important aspect to acknowledge in Model Driven Development is the existing gap between specification-level models and design-level models. At specification level, high-level relationships between elements of the system to be developed are defined. Specifications define 'what' is to be done, normally without specifying 'how' it is to be implemented. Such specifications allow for different designs and implementations from different vendors; all complying to the same specification.

At design level, unambiguous definitions of the elements (and relationships between them) are specified. The design model is an adaptation of the specification model to the selected implementation environment and might contain assumptions related to technological capabilities such as software or communication requirements. Design plays the role of a blueprint for the implementation. A relationship between elements of the design should be established with the code segments of the implementation pieces of code in order to achieve traceability, a desirable property of software engineering [14, 7, 50, 44].

One important feature in MDD is the concept of model executability. It makes use of automatic code generation that bridges the gap between design models and implementations. To do so, precise semantics for each of the constructs and relationships used throughout the design model [91] are needed. We have chosen to use Rational Rose RealTime (ROSE-RT) [33] and its notation to model our SWRadio Design. Our selection is motivated by the following reasons: 1) software radio implementations are real-time systems, 2) the tool supports model executability as our approach requires and, 3) most importantly, because the core constructs used in ROSE-RT have been integrated into the UML 2.0 specification.

In this section, we briefly review the nature and origin of the gap that may exist between specification and design models. Several factors contribute to this gap. We explore only a few and recognize that there is still research to be done in this area. More specifically, we discuss aspects that have had an impact in our case study, which is based on a standard specification model (the SWRadio PIM) and those that are more directly relevant to the use of ROSE-RT for designs. Table 3 presents those aspects and the section where they are addressed within this chapter:

Table 3. Aspects to consider in the gap between specification and design models

Aspect	Section
Representation Language	2.1.1
Design implementing more than one specification	2.1.2
Designs implementing only a subset of a specification	2.1.3
Interface Realization	2.1.4
Procedure-call-based vs. signal-based communication paradigm	2.1.5
Type correspondence	2.1.6
Naming and Role Playing	2.1.7
Mandatory vs. Optional elements	2.1.8
Package Contents	2.1.9
Multi-role players	2.1.10
Specialization	2.1.11

Our discussion of these factors is guided by the following observation: whereas standard specifications are defined in general terms to allow several implementations, in the context of MDA, designs need to capture precise definitions to allow automatic generation of implementations. ROSE-RT adheres to such a conceptual framework.

Solutions to all the challenges presented in this section are proposed in section 3.2.1 of this thesis.

2.1.1 Representation Language

This factor considers the possibility that specification and design models use different modeling languages/metamodels. In our own case study, we work with a specification model written using a subset of the UML 1.5 notation, while our designs use the ROSE-RT notation.

In our opinion, this is perhaps one of the biggest obstacles to establishing direct links between elements in the specification and design models. We consider that the challenge does not directly come from establishing links between elements of the two metamodels, but when this issue is combined with some others aspects. For example, when considering the *Naming and Role Playing* aspect (see section 2.1.7) we will have to establish a relationship between elements of the design that use different names than the elements they are realizing from the specification. If we also have to define different rules for different types of elements (e.g., a protocol *protocolA* in the design needs to be traced back to an interface *interfaceB* in the specification whereas a capsule *capsuleC*

from the design needs to be traced back to a class *classD* in the specification, etc.) then we add an indirection level that tends to make this task more complex.

For model compliance verification, using a specification model that uses a different metamodel than the one of a design model constitutes what we call the *representation language* challenge.

2.1.2 Design implementing more than one specification

This challenge considers the possibility (as is often the case) that a design might implement more than one specification. For example, in the SDR domain the Software Communications Architecture (SCA) specification is a standard that promotes SDR interoperability (see section 2.3.2). An SCA design must comply with the SCA specification and also to other standard specifications such as CORBA, POSIX, and possibly other security standards. Having additional elements in the design not present in the specification may not represent a cause for non-compliance. Such additional elements may correspond to design-decisions taken to support the intended functionality or to realize other specifications. Figure 4 presents a simplistic example of a design model realizing two specifications.

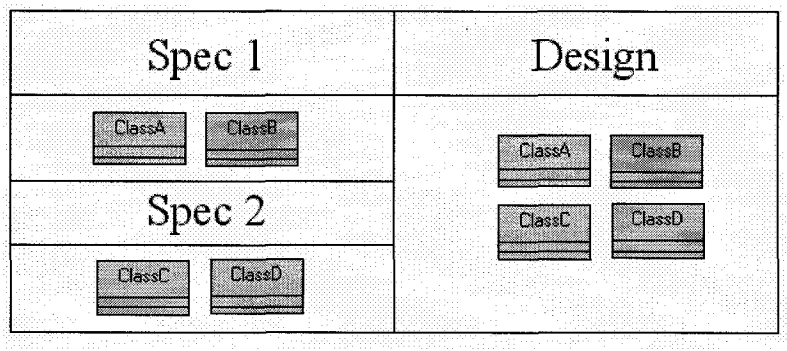


Figure 4. A design implementing two specifications

Figure 4 implies separation of concerns between the two specs. In practice, an element of the design may be associated with elements in different specifications. For model compliance verification, considering a design that realizes more than one

specification constitutes what we call the *design implementing more than one specification* challenge.

2.1.3 Designs implementing only a subset of a specification

This challenge considers a *Design* implementing only a subset of the specification. It is designer driven and gives them the capability to target for compliance only a subset of the specification model elements. Enabling this option allows the designer to check for compliance only the design items that he selects. Figure 5 presents an example of a design model that implements only a subset of a specification.

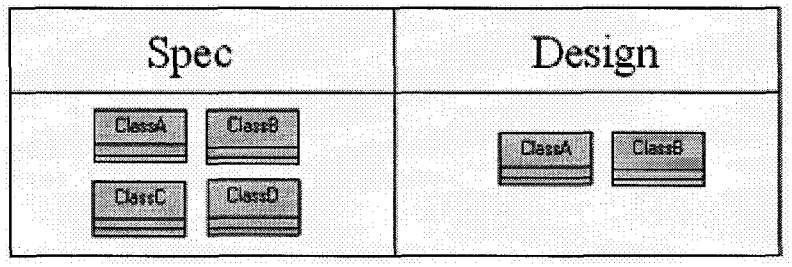


Figure 5. A design implementing a subset of an specification

For model compliance verification, allowing a designer to check for compliance against the subset of the specification model that he is interested in constitutes what we call the *design implementing only a subset of the specification* challenge.

2.1.4 Interface Realization

This challenge considers the fact that an interface realization from the specification may be implemented in different ways in designs that use ports to communicate between active classes[4]. Ports in ROSE-RT as well as in UML 2.0 [76] are bounded by the set of services that a classifier can provide or request through them. The concept of port compatibility requires that every service requested by a port must be provided by a

[4] Active class: A class whose instances may execute its own behavior without requiring method invocation (also referred to as "having its own thread of control"). [76]

connected port, thus establishing a tight coupling relationship between the two connected ports.

Consider the following example: An oversimplified ATM system may be required to read an account value from a card, enable the user to type in his PIN, and display a set of available services to a screen. Although hardware devices (a card reader, a keypad and a display) will perform part of the tasks, software components need to be defined in order to integrate the required functionality. Figure 6 shows an ATM specification model that defines interfaces (and operations) for such devices.

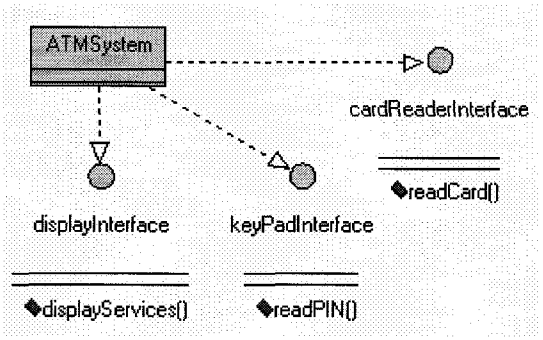


Figure 6. ATMSystem specification model

An implementation capable of processing requests for the operations defined in the interfaces may be deemed compliant with the specification. In the context of MDD, attempting to check for compliance of a design model against a specification model faces different challenges.

Let us consider this *interface realization* aspect with design examples written using ROSE-RT, where: a) a signal-based design approach is used and b) the design modeling language includes ports as the only means of communication between active classes. Figure 7 shows a design model with an *ATMSystem* that contains one port realizing each of the interfaces from Figure 6, whereas Figure 8 illustrates a similar *ATMSystem* but this time containing only one port realizing the three interfaces defined in the specification model from Figure 6.

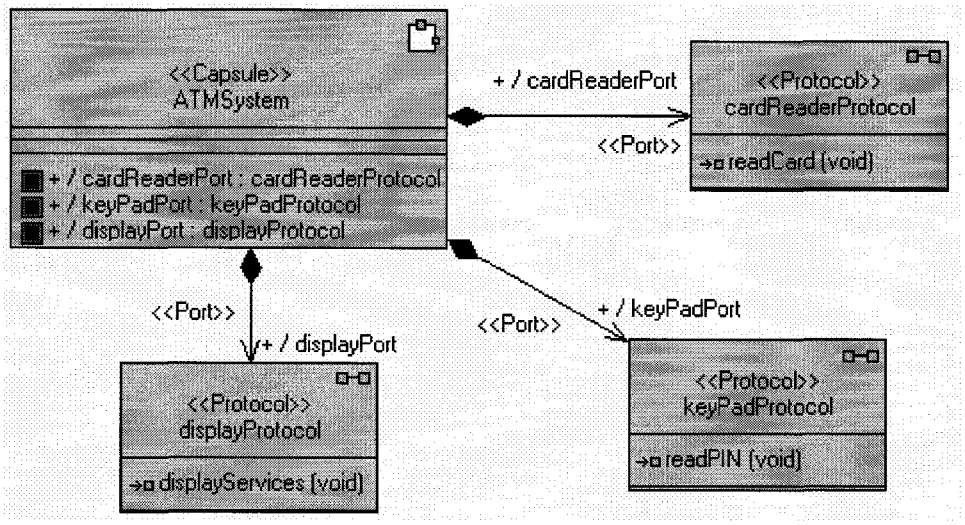


Figure 7. ATMSystem design model with three ports (one port per interface realization)

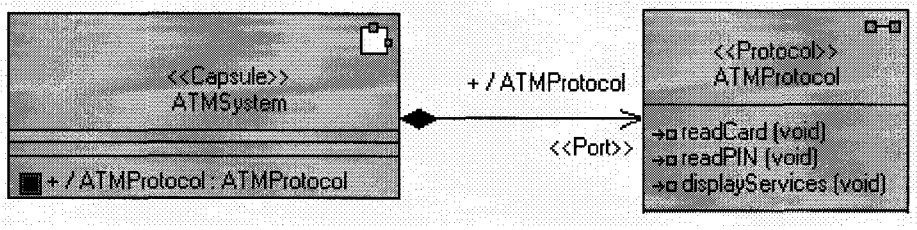


Figure 8. ATMSystem design model with a single port (one port for all interface realizations)

Is one of the two design models compliant while the other is not? Are both of them compliant? Is there a third compliant design model that includes a combination of design models 1 and 2? For model compliance verification, answers to these questions represent what we call the *interface realization* challenge.

2.1.5 Procedure-call-based vs. signal-based communication approaches

Signal-based communications are commonly used in distributed systems. In this section, we present characteristics of the signal-based communication approach and contrast it with procedure-call-based communication. We constrain ourselves to presenting two major differences that impact this research work between the two approaches.

The first major difference we describe relates to the way requests for services (or access to operations and values) are made from one executing object to another. In the case of the procedure-call-based approach, a caller object can invoke the execution of public operations on another object. Such invocation uses a synchronous type of communication. The caller object will block itself during the time the called object executes the invoked operation. Once the called object finishes the execution of the invoked operation, the called object may (or may not) return a value to the caller object. In either case the caller object resumes execution afterwards. In the case of signal-based communications, asynchronous communication is used. The difference from synchronous to asynchronous type of communication means that the caller object does not block itself once the invocation message (signal) is sent to the called object. In the case in which the invoked execution on the called object requires to send a value back to the caller, the called object sends a separate signal to the caller with the return value attached to the signal. The shift translates to two asynchronous signals (one from the caller to the called and another from the called to the caller) in the signal-based approach for every synchronous procedure-call invocation in the procedure-call-based approach.

The second major difference we describe relates to the number of independent values (parameters) that can be sent on an invocation from one object to another. In procedure-call-based communication, the caller can send as many independent values as parameters defined in the operation to be invoked. In the case of the signal-based communication implemented by the ROSE-RT toolset, signals can carry a single piece of data attached to them. The latter means that if we are to send more than a single value, the sending side needs first to encapsulate the operation's parameters into a single data object that will later be attached to the signal. On the receiving side, the receiver needs then to unpack the values if they are to be used during the invoked execution.

For model compliance verification, using a specification/design model that relies on a procedure-call-based communication approach and is checked for compliance against a design/specification model that uses a signal-based communication approach constitutes

what we call the *Procedure-call-based vs. signal-based communication approach* challenge.

2.1.6 Type correspondence

This factor considers the possibility that specification and design models use different names for abstract types when referring to the same primitive data type. For example, while some languages may use the type *Boolean* to define a variable that can hold *true* or *false* values, some others may refer to the same data type simply as *bool.* Figure 9 presents an example of *Spec* and *Design* models with a type correspondence challenge.

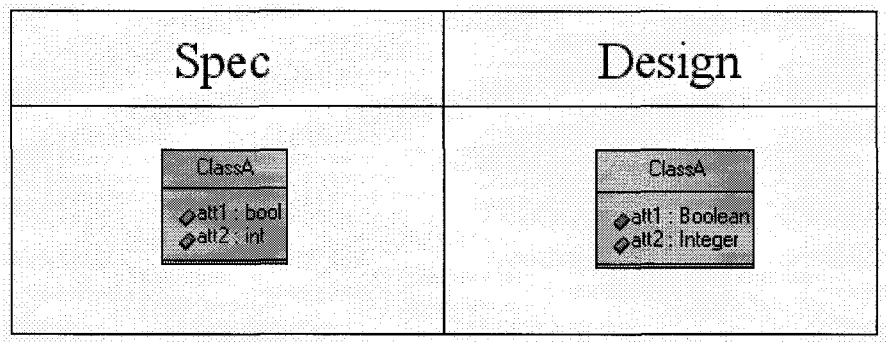


Figure 9. Type correspondence example

For model compliance verification, matching *Boolean* variables from a specification against *bool* variables from a design constitutes what we call the *type correspondence* challenge.

2.1.7 Naming and Role Playing

This factor considers the possibility that an element of the design with name *y* realizes an element of the specification with name *x*. For example, Figure 10 presents a class *DomainManager* from the specification model that is realized by a *DM* class in the design. A search for a *DomainManager* in the design may prove unsuccessful because no such element exists (with the same name).

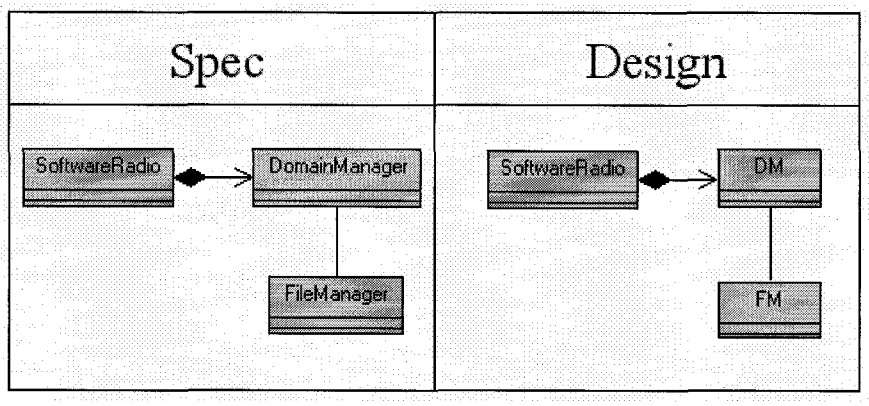


Figure 10. Naming and role playing example

For model compliance verification, matching elements from a design model to elements from a specification model when such elements use different names constitutes what we call the *name and role playing* challenge.

2.1.8 Mandatory vs. Optional elements

This factor considers the possibility that only a subset of elements of the specification model is required (mandatory elements) to be present in the design. Elements from the specification model that are not defined as mandatory are considered optional elements; the designer may decide implementing them or not.

For model compliance verification, establishing compliance where only some elements from the specification model are mandatory (whereas the rest of the elements are considered optional) constitutes what we call the *mandatory vs. optional element* challenge.

2.1.9 Package Contents

This factor considers the possibility that the package structure of the design is different to the one defined in the specification. Packages are used to organize software elements so the system under development is not cluttered with seemingly unrelated elements in the same location. Whether or not to pursue the same package structure is a

challenge that needs to be addressed. Consider for example Figure 11. Is the Design model compliant? We further discuss this issue in section 3.2.1 of this thesis.

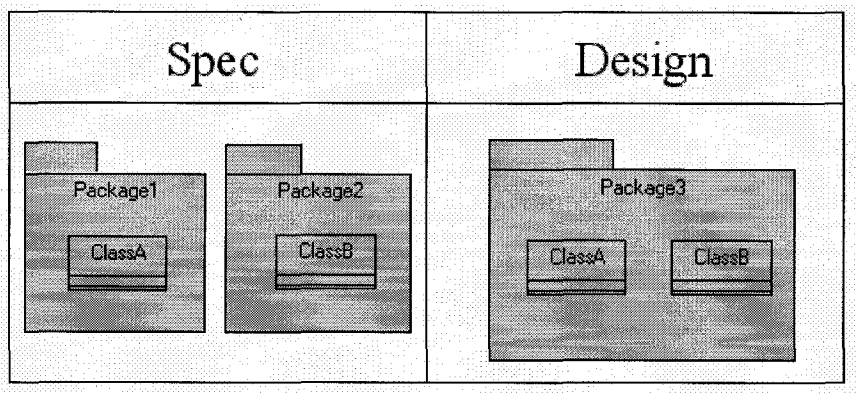


Figure 11. Package contents example

For model compliance verification, enforcing (or not) the same package structure for both specification and design models constitutes what we call the *package contents* challenge.

2.1.10 Multi-role players

Allowing a software element in the Design to play the role of two or more different entities from the specification may respond to design decisions and needs to be considered for model compliance verification. Figure 12 presents a specification model with two classes and two operations each, while the design model presents a single class that implements these operations.

Let us consider that a class *ClassC* in the design realizes classes *ClassA* and *ClassB* from the specification. At first sight, the implications of this problem may not appear to be beyond those presented for the *naming and role playing* challenge. Additional complexity arises if *ClassA* and *ClassB* contain similar properties (like an operation with the same name). Although the property's implementation in *ClassC* may be able to identify which of the intended implementations (from the specification's *ClassA* or *ClassB*) was supposed to handle each request (and act accordingly), a policy needs to be defined for this matter.

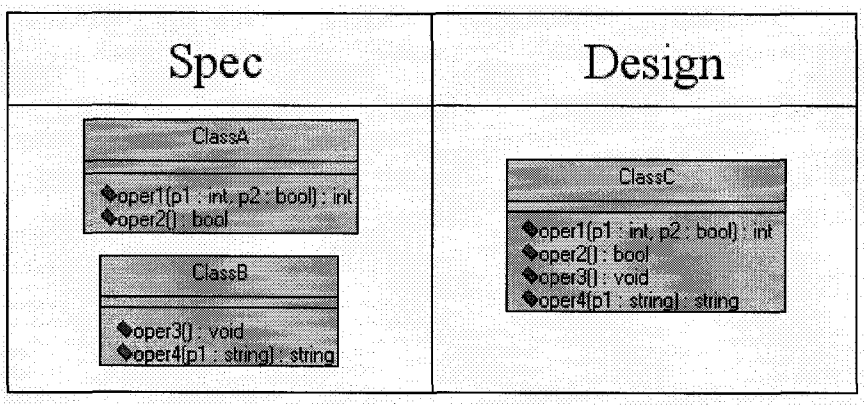


Figure 12. Multi-role player example

For model compliance verification, allowing an element from the design model to realize two or more elements from the specification model constitutes what we call the *multi-role player* challenge.

2.1.11 Specialization

This factor acknowledges the possibility that software elements in the specification might be replaced by specializations of them in the design. As Figure 13 shows, the specification model defines associated elements *ClassA* and *ClassB*, whereas in the design, *ClassA* is associated to *ClassC* as a specialization of *ClassB*. Semantically speaking, *ClassC* should possess the same properties as *ClassB* and thus the association of *ClassA* to *ClassC* in the design should be considered compliant.

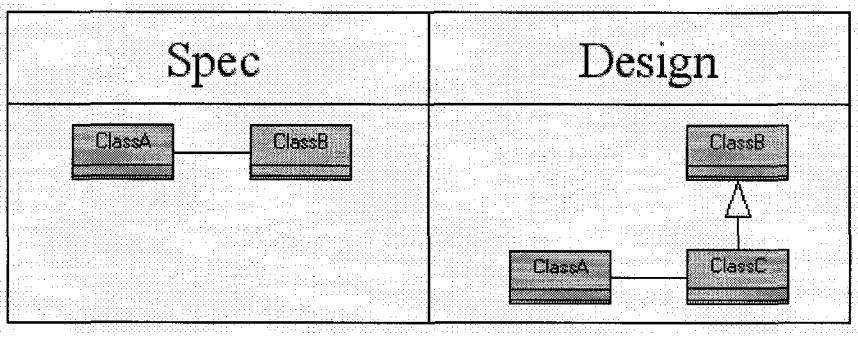


Figure 13. Classifier specialization example

This section refers to classifiers being specialized, but the same rules should apply for properties types and subtypes. In other words, if an attribute of a class is defined as being of *typeA* in the specification model, and that same attribute is found in the design but being of *typeB*, a policy needs to be defined if *typeB* being a subtype of *typeA* should be allowed or not for compliance purposes.

For model compliance verification, allowing an element from the specification model to be replaced by a specialization of it in the design model constitutes what we call the *specialization* challenge.

2.2 Model Transformation and Related International Standards

Models are "a simplification of reality, created in order to better understand the system being created; a semantically closed abstraction of the system" [7]. Model driven development involves the creation and transformation of models to produce software products (executable code, interface definitions, libraries, metadata, etc.). Model transformations have been used for several decades in different software development methodologies. In cases like Top-Down [57, 9, 107], Bottom-Up [27,51,37] and Structured Analysis and Design [96] approaches, model transformation was performed even if it was not explicitly identified. Currently most of software development involves manual transformation processes in most development phases [89,54]. Explicitly defined transformations can be found in approaches based on Correctness Preserving Transformation (CPT) [10,11,15] and OMG's MDA [77]. Transformation definition also plays an important role in systematic development processes like those defined by DeSouza [16] and Bordeleau [8].

The software community is agreeing on a common language for software modeling: the OMG's Unified Modeling Language (UML) [76]). Also and more importantly, there exists agreements on commonly accepted facilities to define modeling languages (OMG's MOF [68]). With such agreements [30, 26], we are witnessing a shift in paradigm from object technology to model technology [4]. Important research is being done to move from an object composition paradigm to a trend where models not only play a pivotal role

in software development, but also allow their automatic transformation to generate implementations. One of the latest agreements in this trend comes in the form of a framework for such model technology: OMG's Model Driven Architecture (MDA) [77]. Within the OMG, one of the first places where transformations were modeled as part of a standard specification was the Common Warehouse Metamodel (CWM) [65]. Because CWM only addresses the transformation from one data format to another, there was still the need to standardize the transformation of models. The need for agreement on the implications and mechanisms for model transformation led to the OMG's MOF's 2.0 Query/Views/Transformations (QVT) Request for Proposal (RFP) [69], which has received revised submissions from several organizations.

In the next subsections we present background information on MDA as well as the above mentioned international standards that support MDA and/or have influenced our research, namely:

- Unified Modeling Language (UML)
- Meta Object Facility (MOF)
- Common Warehouse Metamodel (CWM)
- Query/View/Transformation (QVT)
- Object Constraint Language (OCL)
- Extensible Markup Language (XML)

2.2.1 Model Driven Architecture (MDA)

The OMG promotes MDD in its Model Driven Architecture (MDA) initiative. MDA defines the OMG's vision of the core concepts to perform MDD. In this section, we briefly introduce those concepts.

In [67] a viewpoint on a system is defined as a "technique for abstraction using a selected set of architectural concepts and structuring rules, in order to focus on particular concerns within that system".

MDA defines three viewpoints on a system according to the viewpoint independence (or specificity) of the computational language and infrastructure (platform) in which the system is supposed to execute:

- Computation Independent viewpoint. Defines the environment and requirements of the system. Structural and processing details are not included.
- Platform Independent viewpoint. Defines the operation of a system, hiding the details necessary for a particular platform.
- Platform Specific viewpoint. Defines the operation of a system, including details necessary for a particular platform.

Models realizing viewpoints are called views with regards of the viewpoint they represent. Computation Independent Models (CIM) are views realizing the Computation Independent viewpoint. Platform Independent Models (PIM) are views realizing the Platform Independent viewpoint. Platform Specific Models (PSM) are views realizing the Platform Specific viewpoint. PIMs remain unchanged from one platform to another. A PIM can be defined using general purposes modeling languages or languages specific to the area in which the system will be used. PSMs change from one platform to another.

Figure 14 describes the MDA Metamodel from [77]. It shows the relationships that exist between PIMs and PSMs, and how PIMs and PSMs are described and transformed. Figure 14 defines four types of transformation for PIMs and PSMs:

- Mapping from PIM to PIM: Using PIM Mapping Techniques, a PIM can be mapped into one or many PIMs (all of them independent of the execution infrastructure)
- Mapping from PIM to PSM: Using PIM to PSM Mapping Techniques (not shown in the diagram), a PIM (independent of the execution infrastructure) can be mapped into one or many (not explicitly defined in the figure) PSMs (all of them dependent of the execution infrastructure)
- Refactoring PSM to PIM: Using PSM to PIM Mapping Techniques (not shown in the diagram), a PSM (dependent of the execution infrastructure) can be mapped into one or many (not explicitly defined in the figure) PIMs (all of them independent of the execution infrastructure)

- Mapping from PSM to PSM: Using PSM Mapping Techniques, a PSM can be mapped into one or many PSMs (all of them dependent of the execution infrastructure)

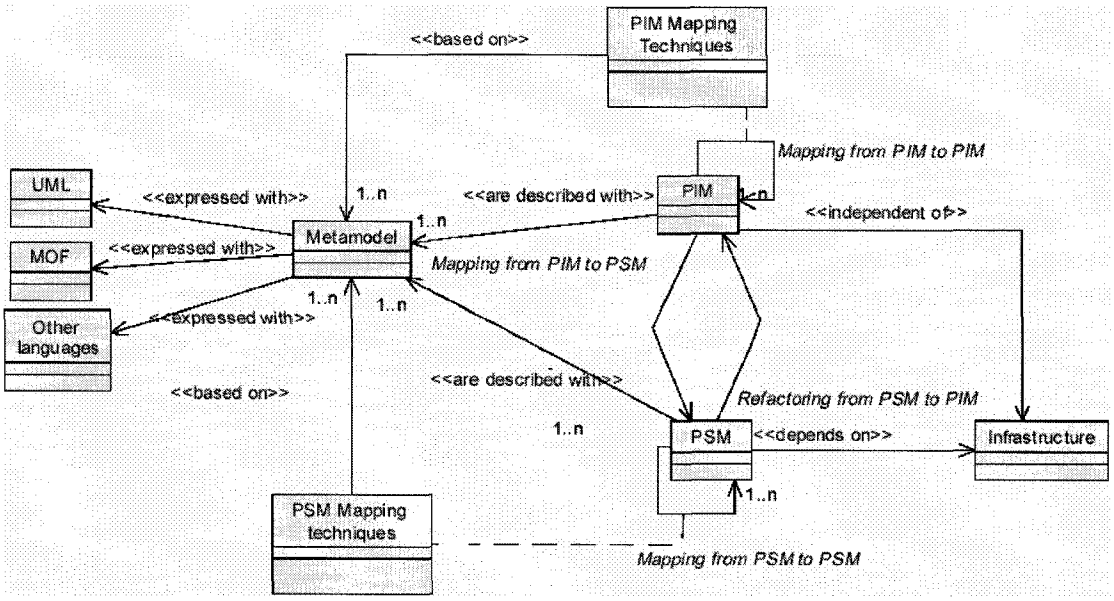


Figure 14. MDA Metamodel Description from [77]

Figure 14 also shows how PIMs and PSMs are described using pre-defined metamodels, which in turn are expressed with UML or other modeling languages. Mapping Techniques are also based on the metamodels used to describe the source and output models. For example, PIM Mapping Techniques are based on the same metamodel used by the PIM being mapped. PSM Mapping Techniques are based on the same metamodel used by the PSM being mapped. Lastly, PIM to PSM Mapping Techniques and PSM to PIM Refactoring Techniques are based on the metamodels used by both PIMs and PSMs being used in the transformation.

2.2.2 Unified Modeling Language (UML)

The Unified Modeling Language (UML) is a graphical modeling language standardized by the Object Management Group (OMG). It is currently the modeling language most often used in the software industry. The current version of UML is UML 1.5 and its first major revision (UML 2.0 [76]) is now at the stage of a Final Adopted

Specification within the OMG. UML 2.0 still needs to undergo through a Finalization Task Force that will eventually produce a UML 2.0 specification to replace the current UML 1.5 version. From here on, we will refer to UML 2.0 simply as UML.

UML provides support for the definition of different aspects of a software system, including different model elements and diagrams. Diagrams are composed of different elements and relationships that exist between them. UML diagrams are divided in two main categories: structural and behavioral diagrams. Figure 15 presents the hierarchy of UML diagrams as defined in [76].

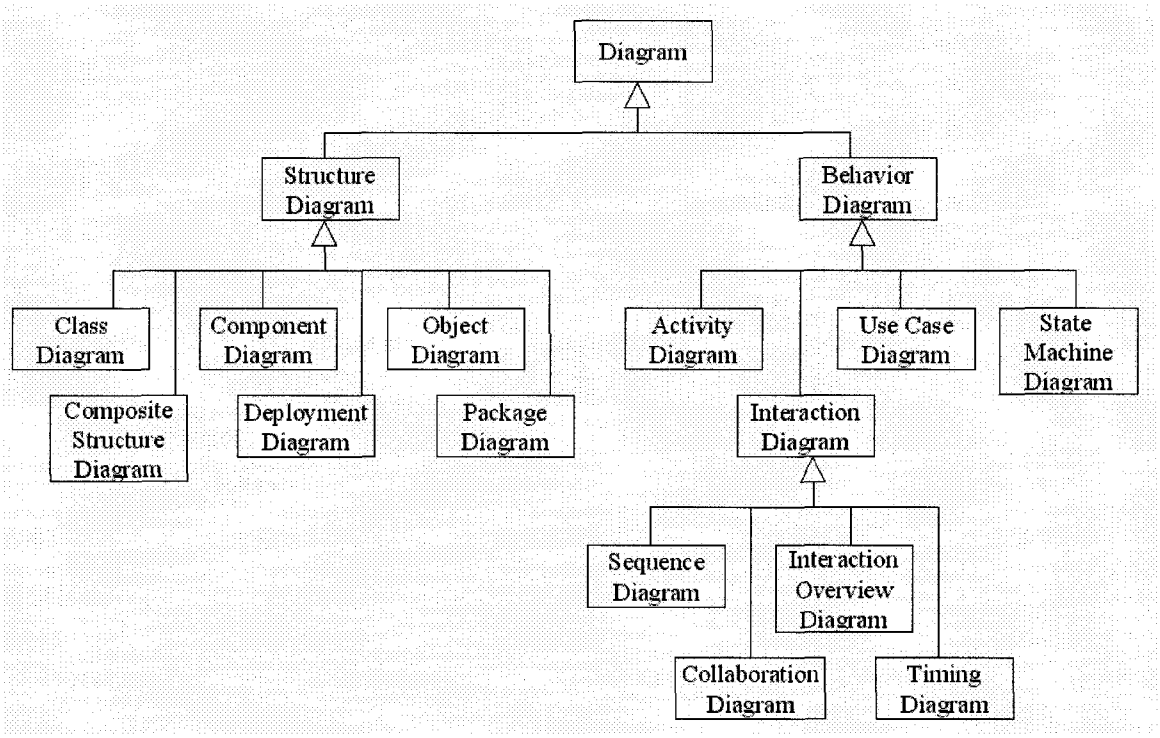


Figure 15. UML 2.0 diagrams from [76]

From the diagrams above, in this background section we only refer to four of them, as the rest were not used in our research[5]:

[5] Elements (and their semantics) are described later in section 4.1.1.

1) Package Diagram: Describes how model elements are organized into packages. It allows for the definition of dependencies (for example package imports and package extensions) between packages.
2) Class Diagram: Describes collections of model elements and their relationships. Represents the static structural aspect of the system under development.
3) Sequence Diagram: Describes interaction between entities/instances of the system, focusing on a time-based definition of messages interchanged. Represents the behavioral aspect of a system.
4) Composite structure diagram. Describes the internal structure of a classifier, which may include interaction points with other parts of the system.

2.2.3 Meta Object Facility (MOF)

OMG's Meta Object Facility (MOF) is a language to define modeling languages. The purpose of the MOF is to regulate and promote common ground for modeling language definitions. Although UML is the most widely used language for software modeling, it was not designed with capabilities to express every kind of software models. Different solutions are considered for those kinds of software models that cannot be expressed using UML only. For example, extensions to UML can be made using UML profiles. Alternatively, non UML-based modeling languages can be defined using the guidelines defined by MOF. The latter implies the possibility to communicate with UML or any other modeling languages.

MOF categorizes models at 4 levels called metalevels. At the bottom of the hierarchy we find the metalevel M0, which is composed of running implementations, objects or data. Metalevel M1 indicates models that can be instantiated into metalevel M0 models. M1 models are mostly blueprints of the M0 running implementations. Models at the M2 metalevel mostly represent modeling languages in which the models of the M1 metalevel are written. M1 metalevel models are then considered instances of M2 metalevel modeling languages. Finally at the top of the hierarchy we find MOF itself as a single model in the M3 metalevel. The metamodel used to define MOF is MOF itself so there is no M4 metalevel. Modeling languages at M2 metalevel (including UML) are considered to be instances of MOF. Table 4 (from [23]) illustrates such relationships:

Table 4. Metalevels described in the Meta Object Facility (MOF) [23]

Metalevel	Description	Elements
M3	MOF, i.e., the set of constructs used to define metamodels.	MOF Class, MOF Association, MOF Attribute, etc.
M2	Metamodels, consisting of instances of M2 metamodel constructs.	UML Class, UML Association, UML Attribute, UML State, UML Activity, etc. CWM Table, CWM Column
M1	Models, consisting of instances of M2 metamodel constructs	Class "Customer", Class "Account" Table "Employee", Table "Vendor", etc.
M0	Objects and data, i.e., instances of M1 model constructs	Customer Jane Smith, Customer Joe Jones, Account 2989, Account 2344, Employee A3949, Vendor 78988, etc.

2.2.4 Common Warehouse Metamodel (CWM)

The Common Warehouse Metamodel (CWM) [65] is an OMG specification that enables data interchange among different data formats (and tools) in distributed heterogeneous environments. CWM is based on three essential industry standards:

- UML - Unified Modeling Language , an OMG modeling standard
- MOF - Meta Object Facility, an OMG metamodeling and metadata repository standard
- XMI - XML Metadata Interchange, an OMG metadata interchange standard.

We include the CWM specification as part of this background chapter as it originally influenced our transformation definitions as well as the OMG's MDA community. Out of the whole CWM specification, we are only interested in the *Transformation* package of it. The package contains metamodel elements for transformations that support the following functions:

- Transformation and data lineage. Transformations of data from a source format into data in a target format. Constraints and operations for data transformation are also defined.
- Transformation grouping and execution. Grouping of transformations into logical units and execution sequences.

- Specialized transformations. Specialized "white box" transformations that are commonly used in data warehousing.

The transformation approach we use in this thesis was originally influenced by the subset of the transformation metamodel that deals with transformation and data lineage. Grouping, execution and specialized transformation defined in the CWM were not considered and therefore not further explained in this background section.

The CWM Transformation Package defines constructs that can be reused when defining data transformation models. Two types of elements are defined (illustrated in Figure 16 and Figure 17):

1. Mappings that specify the type of data (source and target) for the transformation:

- ClassifierMap: takes a classifier from the source data set and creates a new classifier on the target data set.
- ClassifierFeatureMap: takes a classifier/feature from the source data set and creates a new feature/classifier on the target data set. The direction of the transformation (classifier to feature or feature to classifier) is defined through an additional argument (*classifierToFeature*) of type *Boolean*. The default value for the argument is true indicating that the transformation takes a classifier as source and transforms it into a feature as target.
- FeatureMap: takes a feature from the source data set and creates a new feature on the target data set.

2. Transformations that group several Mappings:

- TransformationMap: defines that a mapping of type ClassifierMap will be used as the start for the transformation.

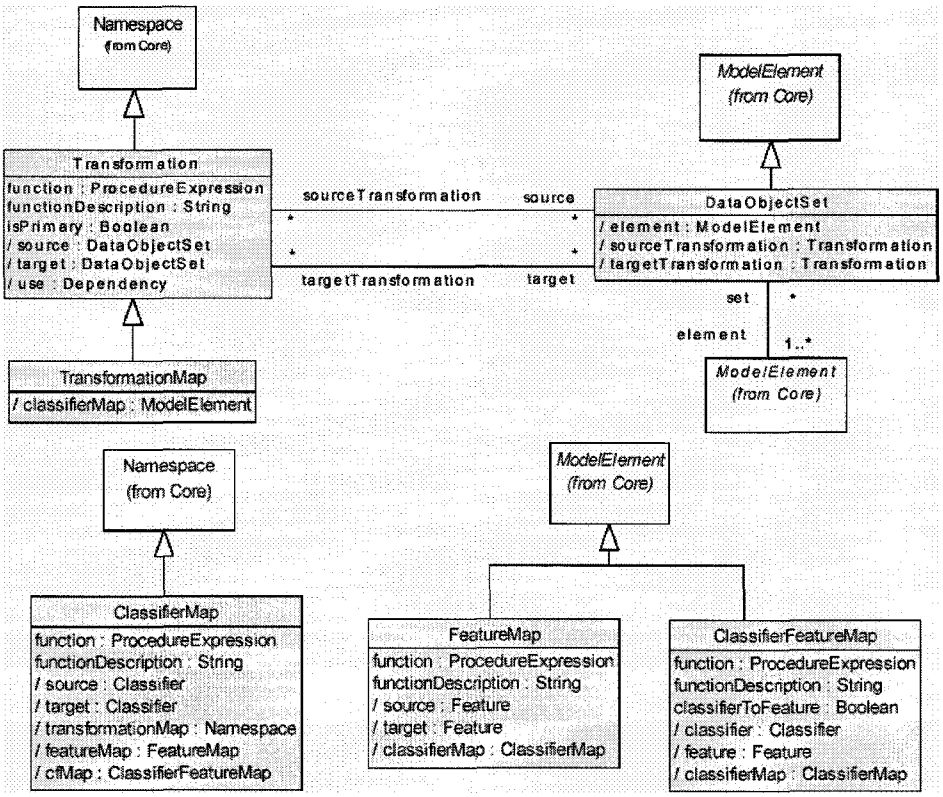


Figure 16. CWM Transformation Package Class definition [65]

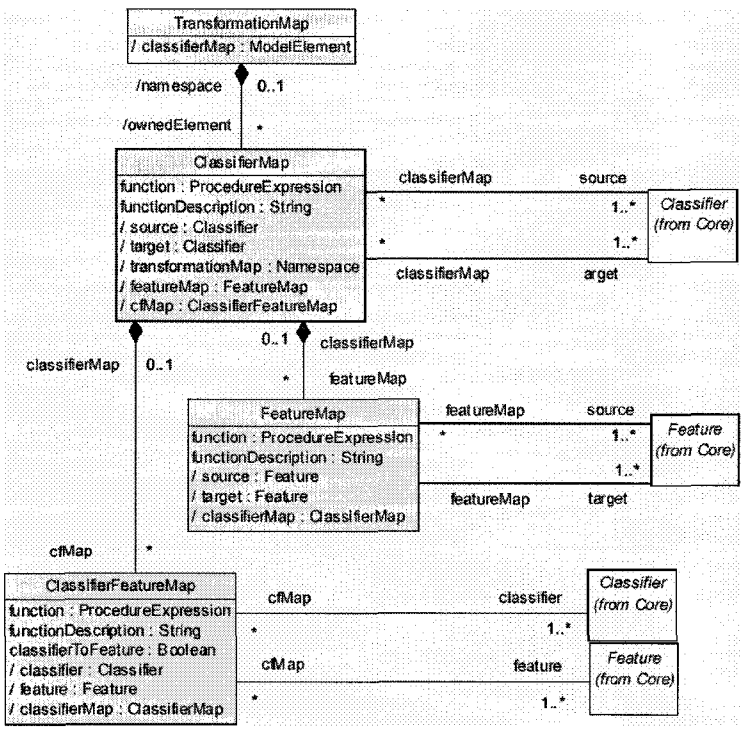


Figure 17. CWM Transformation Package Associations [65]

The transformation is performed by mapping individual elements. If a relational database is to be mapped into a class using the *TransformationMap* defined in Figure 17, a class is created (using a *ClassifierMap*) and the columns are mapped (using a *FeatureMap*) into attributes. Composition relationships between a *ClassifierMap* and a *FeatureMap* provide the required navigability to get access to the elements (in this case features) that compose another element (in this case a classifier).

Compared with our model transformation approach and others submitted to the MOF 2.0 Query/Views/Transformations (QVT) Request for Proposals (RFP) [69], CWM transformations are normally defined at MOF M1 level and executed on MOF M0 data. Model transformation requires definition at MOF M2 level to execute on M1 level models.

Figure 18 presents a common example of a CWM transformation definition (adapted from [23]). The example shows how a relational database table is to be transformed into an XML format document. The collaboration diagram shows on the left side object elements (*POItem*, *pricePerUnit* and *quantity*) as considered instances of the relational data base metamodel elements (*Table* and *Column*). Object elements on the left side of the diagram are considered the source of the transformation. On the right side of Figure 18 the result of the transformation is presented. Elements on this area are considered the target of the transformation. Similar as the source definition, the target definition implies object elements (*POLineItem* and *extendedPrice*) as instances of XML metamodel elements (*ElementType* and *Attribute*). In the center area of the figure the CWM transformation constructs are defined. At the top a *BSBSalesRDBxXML* object (instance of the CWM metamodel element *TransformationMap*) owns the mapping *PurchaseOrderMap* (instance of CWM metamodel element *ClassifierMap*). *PurchaseOrderMap* will create a *POLineItem* in the target definition upon the *POItem* element from the source definition. *PurchaseOrderMap* is also associated with an *ExtendedPriceMap* (instance of CWM metamodel element *FeatureMap*). *ExtendedPriceMap* will create an *extendedPrice* (instance of the XML metamodel element *Attribute*) based upon the contents of the *pricePerUnit* and *quantity* elements

from the relational data base definition. How the *pricePerUnit* and *quantity* elements are transformed into the *extendedPrice* element is defined in a Procedure expression associated with the *ExtendedPriceMap*. The Procedure expression defines that the *ExtendedPriceMap* will take the value of the *pricePerUnit* multiplied by the *quantity* element from the source definition.

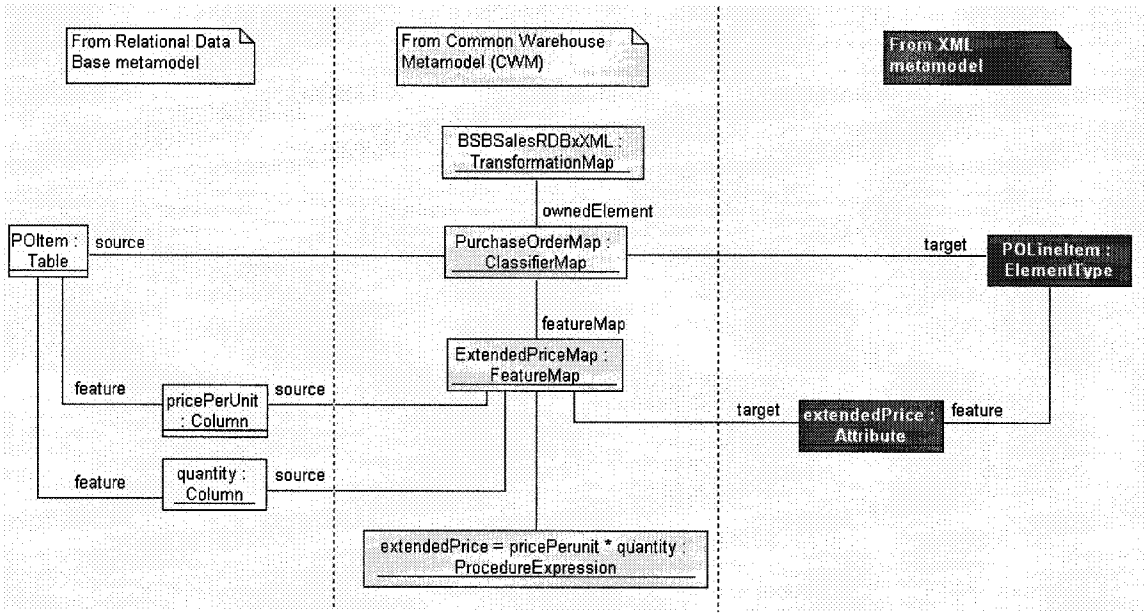


Figure 18. CWM Transformation example

We reiterate that, as shown in Figure 18, transformations in CWM are defined at MOF M1 levels and act on MOF M0 data. We also recall from previous paragraphs that we need to define model transformations at MOF M2 level and apply them in MOF M1 models. The OMG is looking at the lessons learnt with the CWM to define a new standard for model transformation: the OMG's Queries/Views/Transformations (QVT) [69]. We discuss work related to QVT in the next subsection.

2.2.5 Query/View/Transformation (QVT)

The OMG's MOF's 2.0 Query/Views/Transformations (QVT) Request for Proposal (RFP) [69] was issued as an attempt to standardize the way we define and execute model

transformation in the context of the MDA. In this section we will address the more important concepts behind QVT, and present the state of the latest QVT proposals.

2.2.5.1 Query, View and Transformation concepts

The concepts of query, view and transformation are essential in the model driven development vocabulary. We first informally define and explain the concepts, and later use more formal definitions taken from Gardner and Griffin in [28].

Transformations define the elements that must be present on the source model, and the elements that will be present in the target model. The transformation can be defined as unidirectional or bi-directional, although in both cases transformations are executed in one direction at a time. In bi-directional transformations, source and target models exchange roles to become target and source model respectively.

Model transformations are based on finding elements (or set of elements and relationships) on both source and target models. The execution of a simple transformation requires finding an element in the source model, and creating a new element on the target model. A more elaborate transformation will require an update on a target model element based on a source model element. To execute this later transformation, besides the first finding of the source element, a second finding is also required so the element in the target model can be updated. Finding elements (also called matching) in the source and target models is defined as querying in the MDA context.

Informally, a view is a representation of a model (target) that can be directly derived from another model (source). A constraint on views requires that no modifications be made to target models unless source models are modified accordingly. Views are used to abstract away unnecessary detail, and to present the information in a different shape or form so it can be better understood.

Queries, views and transformations are intimately related. A query can be considered as a transformation in which the target model is composed of elements to be found in the

source model with no further actions taken on them. Also a view can be considered to be a specialization of a query, thus making it also a kind of transformation [2].

Based on eight initial submissions for the OMG's QVT RFP [69], Gardner and Griffin define queries, views and transformations [28]:

- "A query is an expression that is evaluated over a model. The result of a query is one or more instances of types defined in the source model, or defined by the query language".
- "A view is a model which is completely derived from another model (the base model). A view cannot be modified separately from the model from which it is derived. Changes to the base model cause corresponding changes to the view. If changes are permitted to the view then they modify the source model directly. The metamodel of the view is typically not the same as the metamodel of the source".
- "A transformation generates a target model from a source model. Transformations may lead to independent or dependent models. In the first case, there is no ongoing relationship between the source and target model once the target has been generated. In the second case, the transformation couples the source model and target model".

2.2.5.2 OMG's MOF 2.0 Query/Views/Transformations RFP and submissions

The OMG's MOF 2.0 Query/Views/Transformations RFP was first released on April 2002. Submissions to MOF 2.0 QVT RFP include different solutions for the following mandatory requirements: query/view/transformation definition at MOF M2 level, between same/similar/different/ metamodels in a deferred/immediate executable fashion which include UML 2.0 extensions and use a definition language independent of the executing technology.

From the original eight initial submissions received by the OMG, five followed up with a first revised submission and only one has been presented as a second revised submission. The latter is the result of collaborative work between five of the original

eight groups that presented initial submissions. Table 5 presents an up-to-date state of the MOF 2.0 QVT RFP. The first reference represents a sequential number arbitrarily assigned to the submission. The second column lists the submitter companies. The third column indicates the latest submission (OMG document) from the submitters. The fourth column indicates whether or not they submitted an initial version. The fifth column indicates whether or not they submitted a first revised submission and/or if they joined efforts with other submitters to strengthen their position. We draw no conclusions on why some submitters did not submit a revised submission.

We remark that the content of the QVT submissions listed below goes well beyond our transformational needs. The reason is simple: QVT submissions are supposed to capture a wide variety of model transformation requirements while our research work is best represented as a specific application of some of those requirements. In essence, any of the QVT submissions could be considered as the metamodel for our transformation definition.

We consider that the value of this thesis work does not come from defining the representation of model transformations (i.e., a model transformation language), but from its application towards what we call compliance verification of models. It is by making use of model transformation that we intend to verify the compliance of a SWRadio design with respect to its SWRadio specification.

As the QVT submissions are still subject to modifications and will undergo a detailed evaluation from the appropriate OMG committees, we choose to keep using our own CWM-influenced transformation definition format. We consider that our approach complies with a subset of the mandatory requirements of the MOF 2.0 QVT RFP: Query/transformation definition at MOF M2 level, between different metamodels in a deferred executable fashion: it includes UML 2.0 extensions and uses a definition language independent of the executing technology. We further elaborate in section 3.1.1. The evaluation of the QVT submissions is considered out of the scope of this thesis.

Table 5. OMG's QVT RFP submissions.

	Submitters	Latest Submission	Initial Submission	First Revised Submission	Second Revised Submission
1	DSTC IBM CBOP	ad/04-01-06 [22]	Yes	Yes	Yes... Merged with 4
2	Alcatel Softeam Thales TNI-Valiosys Codagen Technologies Corp	Ad/03-08-05 [2]	Yes	Yes	Merged with 4
3	Compuware Corporation SUN Microsystems	Ad/03-08-07 [13]	Yes	Yes	No
4	QVT Partners: Tata Consultancy Services Artisan Software Colorado State University Kinetium King's College London University of York	Ad/03-08-08 [99]	Yes	Yes	Yes
5	Interactive Objects Software GmbH Project Technology	Ad/03-08-11 [34]	Yes	Yes	No
6	Kennedy Carter Ltd	Ad/03-03-11 [43]	Yes	No	Merged with 4
7	Adaptive	Ad/03-02-09 [1]	Yes	No	Merged with 4
8	Codagen Technologies Corp.	Ad/03-03-23 [12]	Yes	Merged with 2	No

2.2.5.3 A QVT realization in progress: The Model Transformation Language (MTL)

The Model Transformation Language (MTL) [102] is the corner stone of an advanced proposal of a model transformation framework called Umlaut [37]. Led by Jean-Marc Jézequél, MTL was developed within the context of QVT by the Triskell research group [101] at the Institut de Recherce en Informatique et Systèmes Aléatoires (IRISA) [36]. Originally focused in transforming UML models, Umlaut has evolved to allow transformation of models written in other modeling languages using MTL. Key properties of MTL are its executability and facilities to transform Domain Specific Languages (DSL) and generic model transformation languages into MTL.

Umlaut includes development tools that support MTL. The tools are defined in a three-tier architecture [103] that provides independence of transformation language, transformation models and transformation execution from the models being transformed. Transformation models are defined using MTL in an Integrated Development Environment (IDE) called MTL CASE. A MTL engine that extracts models from different modeling tools executes model transformations.

2.2.6 Object Constraint Language (OCL)

The Object Constraint Language (OCL) is a formal language used to describe expressions on UML models [75]. OCL is a pure specification language and not a programming language. Alternatives to OCL include the definitions of constraints in natural language (which normally lead to ambiguities or imprecise constraints definitions), or the use of more formal languages (that usually require strong mathematical background for its application). OCL's origin lies in the IBM Insurance division and it was developed as a business modeling language rooted in the Syntropy method [75].

The principal properties of OCL Expressions are:

- used to specify invariant conditions or queries over objects described in a model
- programming language independent
- evaluation of an OCL Expression do not alter the state of the executing system in which the OCL Expression is being evaluated
- by definition not directly executable
- must comply with the type conformance rules of OCL (i.e., it is not possible to compare an integer with a string)
- the evaluation of an OCL expression is instantaneous.

The following is an example of an OCL expression:

context p : Person inv:

p.age > 18

where the context specifies the element in which the invariant is to be applied. In this example the context is defined to be *p* of type *Person*. The invariant specifies that the value of the *age* property of *p* at the moment of the evaluation must be an integer value greater than 18. This specific constraint can be used when modeling the application for a driver's license that specifies that the applicant must be 19 years or older to apply.

As the use of OCL in this thesis is at a very basic level, we do not further elaborate about OCL.

2.2.7 Extensible Markup Language (XML)

The Extensible Markup Language (XML) is a standard to define rules for data formatting with data interchange or data storage purposes. XML has its origins in the Generalized Markup Language (GML), a project developed by IBM in the early 1970s. During the late 1970s and early 1980s the American National Standards Institute (ANSI) developed a standard text-description language based upon GML. The result of such work was the Standard Generalized Markup Language (SGML) that was quickly adopted by the U.S. Department of Defense and the U.S. Internal Revenue Service [87]. The complexity of SGML resulted in it being used only by large organizations. In the early 1990s the Hypertext Markup Language (HTML) was developed and was quickly adopted as the medium to present and format content to be exposed to the World Wide Web. As HMTL was conceived with a predefine set of tags to format the content of an HTML document, more flexibility was required. XML filled the gap by being a subset of SGML and allowing full interoperability with both SGML and HTML. The goal of XML is to enable generic SGML to be served, received, and processed on the Web in the way that is now possible with HTML [106].

Because the use of XML in this thesis is very limited, we only need to introduce the notion of tags, which are the simplest form of XML definition.

In XML, the content of a document is enclosed within markup symbols known as tags. Tags are placed before (and commonly after) the section of the document that wants to be marked. Tags are defined by placing the tag's name enclosed between angle brackets (< >). The tag itself is known as an element and the text placed between tags is known to be the content of the element. Elements can be defined inside other elements.

A valid definition of an element inside another element concludes this background section on XML:

<element> <element1>content of element1 </element1></element>

2.3 Software Defined Radio

In this section, we introduce the Software Defined Radio (SDR), which provides a context for our case study. We first describe the SDR concept. We then survey some of the proposals towards the standardization of SDRs. We conclude this section with an overview of the Software Radio (SWRadio) PIM developed by the OMG Software-based Communication (SBC) Domain Task Force (DTF). This SWRadio PIM will act as the starting point for the specification model we use in our work.

2.3.1 Software Defined Radio Concept

Software Radios (also called Software Defined Radios) are an emerging technology with no single definition yet for such communication devices. Joseph Mitola III coined the software radio term back in 1991. He defines them as "Radios whose channel modulation waveforms are defined in software" [58].

The notion of a Software Defined Radio defines a communication device that uses generic components that are configurable by software, replacing the application-specific

integrated circuits used in non-SDR devices. An SDR is a wireless communication system in which the particular communication and transmission characteristics are realized through specialized software running on flexible signal processing hardware as opposed to the traditional approach of using specialized hardware for the same functions [79]. "The hardware of a Software Defined Radio consists of Digital Signal Processors (DSPs), Field Programmable Gate Arrays (FPGAs), General Purpose Processors (GPPs) and other hardware components. The behavior and characteristic of each of these hardware-building blocks is controlled by a dedicated software component. The collection of these components defines the communication and transmission characteristics of the radio and is therefore called Waveform Application, or just Waveform. Examples of waveforms include ATC (air traffic control), police, cell phone, and military communications" [79].

2.3.2 Towards the standardization of Software Defined Radios

Several institutions and international organizations are currently working towards standardization of SDRs interoperability. The Software Communications Architecture (SCA) specification [41] is, in our opinion, the most noticeable of these initiatives. The SCA was originally developed by the Modular Software-programmable Radio Consortium (MSRC)[6] [60] for the Joint Tactical Radio System (JTRS) Joint Program Office (JPO) [39] as a request of the U.S. Department of Defense.

Another noticeable contribution towards the SDR standardization has been introduced by the Software Defined Radio Forum (SDRF) [97], which defined a Software Radio Architecture as the integration of the SCA to the SDRF community.

There is also the work of the Object Management Group Software-Based Communication Domain Task Force [78] that is contributing towards standardization of SDRs. The objective of the SBC DTF is to define a specification of a new generation of

[6] The Modular Software-programmable Radio Consortium (MSRC) is formed by Raytheon, ITT Industries, Rockwell Collins and BAE SYSTEMS.

radios that can be reconfigured at run-time. This new generation of radios aims to support new critical needs (e.g., several waveforms inside the same box, easy bug fixing, radio core functions reconfigurability, support for digitized, IP-based, data transmissions, and improved security). Following the OMG's Model Driven Architecture approach, the SBC defines its specification in the form of a Unified Modeling Language Platform Independent Model. The Software-Based Communication DTF has submitted for approval SWRadio PIM to be defined as an OMG Software Defined Radio standard specification. We briefly introduce the Software Radio (SWRadio) PIM in the next section.

2.3.3 Software Radio Platform Independent Model

The OMG's SBC SWRadio PIM describes high-level relationships between domain concepts that correspond to the main entities that compose the Software Radio architecture and the required interface of those concepts. It uses only conventional UML 1.5 notation, i.e., mostly class diagrams, with some use cases, sequence diagrams, and very few simple state machines.

The SWRadio PIM is defined in a hierarchy of 43 packages. In these packages, 221 classes and 19 interfaces are defined. Classes and interfaces are composed of 112 operation definitions, which together define 84 parameters. Classes and interfaces also define 271 attributes. In terms of relationships, the SWRadio PIM defines 22 simple associations, 20 aggregations, 102 compositions and 60 generalization relationships between classes. There are also 18 realize relationships between classes and interfaces.

Due to the size of the SWRadio PIM and the scope of this thesis, in this section we only introduce a subset of the SWRadio PIM. The first part of this section introduces the top-level packages that constitute the SWRadio PIM. The second part specifically targets the Core Framework control package as the subset of the SWRadio PIM we reference the most in our case study.

2.3.3.1 SWRadio Top Level Packages

The SWRadio PIM is grouped into 43 packages, some of them being contained in other packages. Figure 19 presents the SWRadio PIM package dependencies.

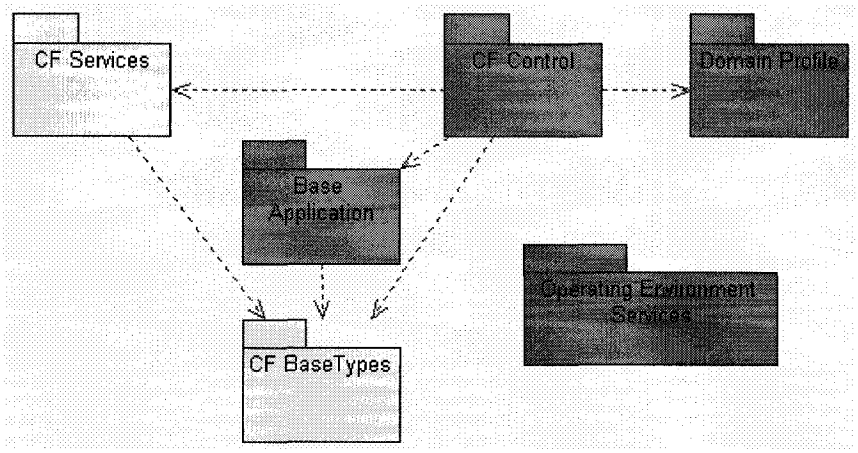


Figure 19. SWRadio PIM Package dependencies.

The following packages represent the top-level containers of the SWRadio PIM:

- CF Services: Defines the services offered by the SWRadio Core Framework (CF). Among them we find File, Naming, Event and Security services related definitions.
- CF Control: Defines the CF control classes within a Domain. CF Control classes are divided into Domain Management, Device, and Device Management classes.
- Domain Profile: Defines properties and contents of the Domain. The package is composed of the following packages: Assembly Deployment Specification, Device Configuration Deployment Specification, Domain Manager Configuration Specification, Embedded Component Assembly Deployment Specification, Embedded Component Deployment Specification, Embedded Component Specification, HW Device Specification, Property Types Specification.
- Base Application: Defines a set of interfaces that can be used by software applications. Interfaces defined in this package are Port, LifeCycle, TestableObject, PropertySet, PortSupplier, ResourceFactory, and Resource.

- Base Types: Defines the CF underlying types used while in the deployment and configuration of applications.
- Operating Environment Services: Yet to be defined in the SWRadio PIM.

2.3.3.2 SWRadio Core Framework Control Package

In this thesis, we use a subset of the SWRadio PIM to test and validate our approach. We mainly work with the CF Control package. In this section, we first describe the elements that constitute the CF Control package and later describe a high level overview of the relationships that exists between them. We have deliberately excluded from this section the UML class definition and other low-level relationships of each of the SWRadio classes described below.

The following classes are defined in the CF Control Package:

- DomainManager: Control and configuration of the system domain.
- DeviceManager: Manages a set of logical devices and services.
- Device: Type of Resource within the domain. Device extends the Resource interface and thus inherits the Resource requirements. Device defines additional capabilities and attributes for logical Devices in the domain.
- LoadableDevice: Extends Device by adding loading and unloading capabilities to the behavior of a Device.
- ExecutableDevice: Extends LoadableDevice by adding executing and terminating capabilities to the behavior of a Device.
- AggregateDevice: Provides aggregate behavior to add and remove Devices from an aggregate Device.
- Application: Provides mechanisms for the control, configuration, and status of an instantiated application in the domain.
- ApplicationFactory: provides the mechanism to request the creation of a specific type of *Application* in the domain.

Figure 20 presents the CF Control Overview class diagram. At the top of the diagram the DomainManager is shown. A DomainManager can be composed of zero or many running Applications while serving as an Application directory for the Applications themselves. The DomainManager is also associated with zero or many DeviceManagers and with zero or many Devices. The DeviceManagers act as an information provider for the DomainManager that plays the role of the registrar in the association relationship. Devices playing the role of products and registrants are associated with one DeviceManager in the role of creator and registrar respectively. Devices allocate and deallocate memory upon requests from the ApplicationFactory and Applications. Such relationship is shown using two association relationships from the Device to the ApplicationFactory and Application classes in the diagram. Applications are the product of ApplicationFactories that act as creators of Applications. LoadableDevices and ExecutableDevices are specializations of Devices and use the ApplicationFactory as the Application Configure Manager. Applications are loaded and executed in Loadable and Executable Devices.

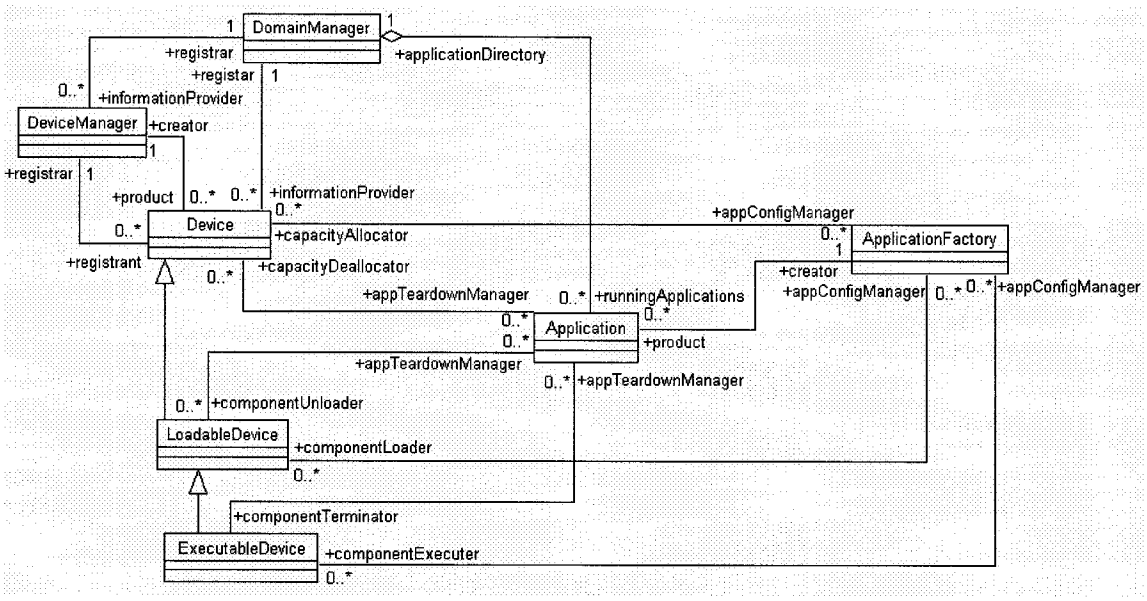


Figure 20. SWRadio Core Framework Control Overview class diagram.

2.4 Verification

Deutsch in [17] defines software verification as an activity that assures that the results of successive steps in software development life cycle correctly embrace the intentions of the previous steps. In this section, we elaborate on background information related to verification in different areas of interest for our research. We first address literature related formal software verification. We then present verification of model-to-model transformations in MDD. We finally address information about the current technology used to verify compliance of software radios.

2.4.1 Formal Methods

Formal Methods is the umbrella term that computer scientists use to refer to the set of mathematic-based techniques, notations and tools, that are used in the development and verification of software artifacts [87]. Formal methods in software development require the use of formal languages as well as formal reasoning. The body of formal techniques is supported by precise mathematics and powerful analysis tools. Most often they imply rigorous and efficient mechanisms for system modeling, synthesis and analysis.

Formal methods in software development have been used (and still are) for mainly two purposes [87]:

1) Stepwise derivation of programs, rooted in Dijkstra's program transformations [18] and Partsch's program adaptation, modification and synthesis [81].
2) Formal verification of computer systems, which involves the definition of formal arguments of correctness, based on mathematical principles.

Generally, formal software verification usually follows the application of formal techniques for stepwise derivation of programs. Less frequent (but also applied) is stepwise derivation of programs without results being formally verified [95]. A key property of both cases is the presence of a formal specification [87]. In the case of stepwise derivation of programs, the formal specification is used as the source artifact from which the program is to be derived. In the case of formal verification, the software artifact is verified against such formal specification.

The use of formal methods in software development has recently gained terrain in industrial applications [31, 59, 61, 95]. Although we acknowledge the possible use of formal methods for the creation of Software Radio Design models, as mentioned before, the topic lies beyond the scope of this thesis and we do not further discuss the issue[7].

With respect to formal verification there are several reasons why we discard its use in this thesis. First and foremost because our Software Radio standard specification lacks the formality required for such purpose. Our work was originally motivated by a domain-specific problem: establishing compliance of software designs against a Software Radio specification. We remind the reader that our SWRadio PIM is written using conventional UML, which again, lacks the formality required to apply mathematical techniques in UML models [3, 104]. A second argument not to pursue formal verification in this thesis, relates to our aim to make our proposed solution accessible to practicing software developers without requiring additional knowledge about more comprehensive formal verification approaches. Finally, there is also the cost associated with formalizing the SWRadio specification that we consider decreases substantially the feasibility of the verification process.

2.4.2 Verification of Model-to-model Transformation in MDD

At the beginning of this thesis work, not much literature was available on the topic of verification of model-to-model transformations. Little explicit material was found related to model transformation, while almost none make explicit reference to verification of such transformations. We came to the conclusion that even though model transformation/verification was not new in the industry, it was often implemented in an ad hoc manner with a wide range of different solutions. Of the few available model transformation/verification works prior to the QVT RFP, we found the following similarities:

1. Identification of source and target models' metamodels. Defining transformations/verifications to act on models required targeting the languages in

[7] We remind the reader that we are interested in verifying that a design model complies with a

which those models were written (also known as metamodels). Related work can be found in Mellor and Balcer [52], Bézivin [5], Milicev [55, 56], Selonen et at. [94], de Miguel et al. [54], Lemesle [48], Domínguez and Zapata [19, 20, 21] and Kovse et al. [46].

2. Representation language for model transformation. Besides identifying the elements that constitute the languages (metamodels) being used to define the source and target models, the questions of the notation and how to represent the metamodels themselves were discussed. We consider the proposals we found in this direction were implementation biased and the most frequent solution was to define metamodels through an XMI representation. Related work can be found in Bézivin [5], de Miguel et al. [54], Porres [85, 86], Kovse et al. [46], Warner [105], Leblanc and Merle [47].

   An alternative to XMI representation was proposed by Jézéquel et al. in [98]. They propose to use UML Action Semantics (AS) as the language to represent model transformations. They argue the virtue of full integration of AS with UML, as well as its application not only to UML models, but to the UML metamodel itself. The key principle behind the latter is the self defining property of UML[8]. In other words, the UML metamodel is defined using UML elements, thus making it also a UML model.

3. Language used for rule definitions in model transformation. The aim was to identify a rule definition language that could unambiguously extract (or match) a subset of a model. The most frequent selection we found was OCL based rule definitions. Related work include Jézéquel [98, 84], Bézivin [5], Gerber et al. [29], Peltier et al. [82, 83], Miguel et al. [54].

The works presented in the previous paragraphs are individual (sometimes groups of individuals) contributions towards model transformation in general. As the topic became an essential piece for the application and implementation of MDA, some of the same researchers (now representing their companies as OMG members) have submitted

specification model, and not in how to create or derive a design model itself.
[8] UML is defined using the OMG's Meta Object Facility (MOF) which in turn is defined in terms of UML. UML and MOF share the same metamodel infrastructure

contributions for the MOF 2.0 QVT RFP (Bézivin, Vojtisek, Jézéquel [2], Gerber, Lawley, Raymond, and Steel [22] and Mellor, Uhl [34]). For more on these submissions see section 2.2.5.

2.4.3 Compliance of Software Defined Radios

With respect to compliance verification of SDRs, work is being done to define a compliance-testing framework to verify the compliance of SDR products with respect to their common specification. The current state of this framework is best understood by studying the SCA. In the Architecture Compliance section of the SCA we find the following definition: "Products submitted as 'SCA-Compliant' will be evaluated for compliance in accordance with the test methods and procedures established per section 7.2" [41]. Although section 7.2 of the SCA has yet to be defined, the JTRS JPO has established the JTRS Technology Laboratory (JTel) [40] whose mission is to "provide JTRS waveform acceptance and SCA-compliance recommendations to the JTRS JPO".

We can only emphasize the importance of this effort. The JTRS Mission Description and Budget Item Justification [42] includes a budget estimate for half a billion US dollars to be spent over the 2003-2009 period. Almost 10% of the actual and estimated resources allocated for the 2003-2005 period will go to SCA Compliance testing related activities as shown in Table 6.

Table 6. Estimates for JTRS and SCA compliance testing activities (in thousands of dollars)

Cost	FY 2003 Actual	FY 2004 Estimate	FY 2005 Estimate	FY 2006 Estimate	FY 2007 Estimate	FY 2008 Estimate	FY 2009 Estimate
JTRS	$ 62,892	$ 133,293	$ 121,400	$ 71, 221	$ 57,233	$ 28,573	$ 27,073
SCA Compliance Testing	$ 10,396	$ 9,414	$ 11,636				

In this thesis, we work in the context of the OMG SWRadio PIM. To our knowledge there is no published or ongoing work within the OMG related to compliance verification of SDRs. We hope our research will be integrated with the Software-Based Communication DTF efforts towards SDRs standardization.

Chapter 3. Compliance Verification Approach with Automated Support

In this chapter, we describe at length one of our main contributions: our proposed two-step approach as refined in Figure 21. At the core of the contribution is the introduction of an intermediate model that is semantically equivalent to the specification model but expressed in the same language as the design model. We also consider the possibility and usefulness of automation throughout this approach. By building prototype tools, we speed up the application of the approach and provide a proof of concept for it. Prototype tools are explained in detail in section 3.3.

Following the organization of section 1.4, we first address in section 3.1 the *Spec* to *tSpec* transformation-compliance activities. Then, we tackle step 2, that is *tSpec* to *Design* realization/compliance, in section 3.2.

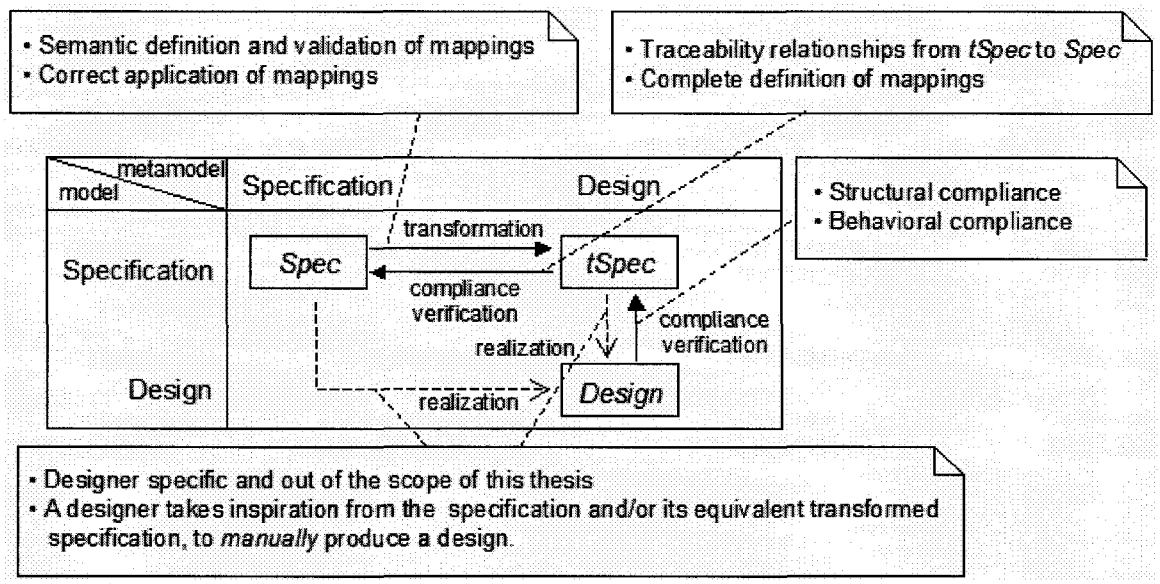


Figure 21. Detailed two-step approach for Transformation/Compliance

Figure 22 presents an activity diagram with a more detailed view of the activities of our proposed approach. The intention is to give the reader a high level view of how the different sections in this chapter are related to each other.

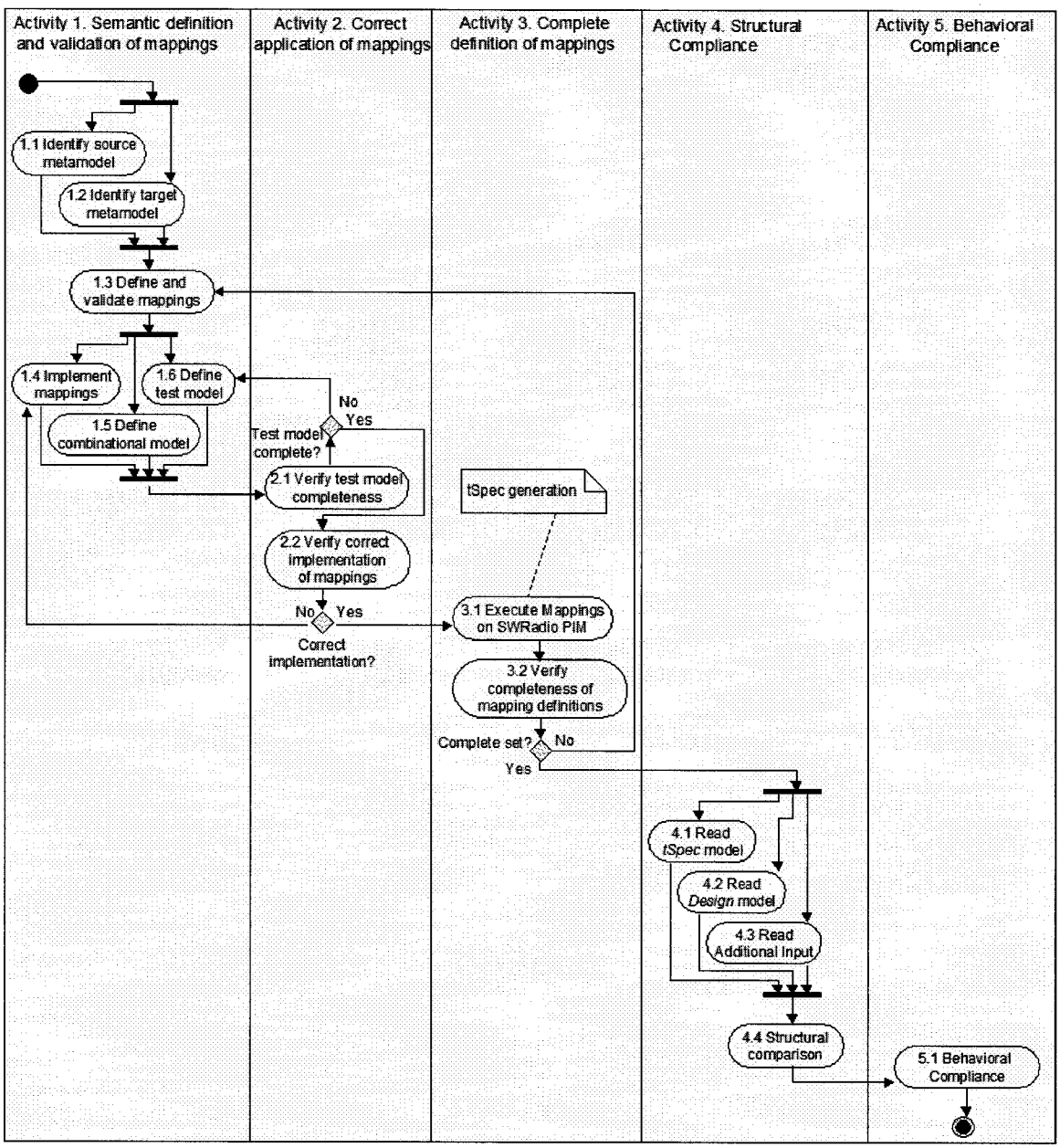


Figure 22. Detailed two-step approach for Transformation/Compliance

In activity 1 from Figure 22, we consider the identification of source (activity 1.1) and target (activity 1.2) metamodels as the first activities to perform in our approach. We

continue then by defining mappings and validating them (activity 1.3). After that, we proceed to implement the mappings (activity 1.4), define combinational models (activity 1.5) and define test models (activity 1.6) before we proceed to verify the correct implementation of the mappings in activity 2. Within activity 2, we first verify that the test models being used to test the implementation of the mappings are complete (activity 2.1). If not, we return to activity 1.6 to update the test models. Once the testing models are complete, we proceed to verify that the implementations produce the expected results as defined in activity 1.5. If not, we go back to activity 1.4 and modify implementation until it achieves the expected results. Once activity 2 has been carried out, we continue with the execution of the mappings, but this time using the actual SWRadio PIM (activity 3.1) instead of using test models. That takes us into activity three of the approach. We verify then that every element in the SWRadio PIM has been used as originator of an element of the tSpec model (activity 3.2). If not, we go back to update or define new mappings in activity 1.3. Once we have a complete set of mappings we proceed with structural compliance in which a separate tool will first parse the tSpec (activity 4.1) and Design (activity 4.2) models, read additional input if provided (activity 4.3) and proceed with structural comparison (activity 4.4). We end up the application of our compliance verification approach with activity 5.1 related to behavioral compliance. In the next sections we describe in details each of the activities presented in Figure 22.

3.1 Spec to tSpec Transformation and Compliance

As previously explained, activities of Step 1 are highly related to the representation language or metamodels used in defining *Spec* and *Design* models. Recall that in this first step we translate the specification model into a semantically equivalent model, the *tSpec*, expressed using the same language as the intended design.

Let us then define semantic equivalence between two models: We say that a model *B* is semantically equivalent to a model *A* when the following conditions are met:

1) Model *A* is written using metamodel *mA* and model *B* is written using metamodel *mB*.

2) There exists a set of mappings *M* that takes elements from metamodel *mA* and transform them into elements of metamodel *mB*.
3) Within the set of mappings *M*, there exists at least one mapping that applies to each element defined in metamodel *mA*[9].
4) Implementations of mappings have been proved to produce output as expected.
5) When mappings are executed on model *A*, all the elements contained in model *A* are used as originators of elements of model *B*.
6) Model *B* is automatically generated after execution of mappings *M* on model *A*.

Figure 23 shows two non-semantically equivalent models: *Spec* and *tSpec*. Dotted arrows represent mappings that take elements from the *Spec* model and transform them into elements of the *tSpec* model. Mappings define one to one relationships (e.g., *A* from the *Spec* is the originator of *i* in the *tSpec*), many to one relationships (*C* and *D* from the *Spec* are the originators of *iii* in the *tSpec*) and one to many relationships (*G* in the *Spec* is the originator of *v*, *vi* and *vii* in the *tSpec*). When considering the conditions defined above, we declare that the two models are not semantically equivalent because condition four has not been met: namely, element *E* from the *Spec* is not represented (or was not used as originator of an element) in the *tSpec*.

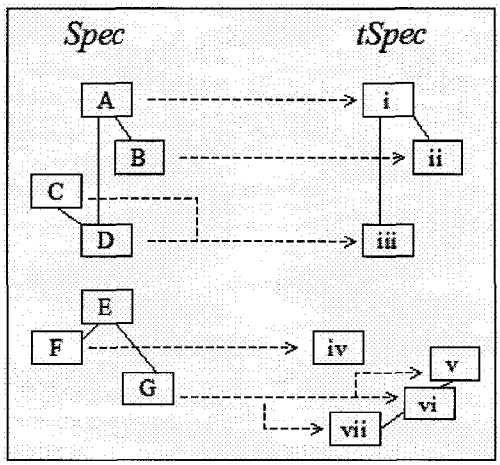


Figure 23. *Spec* elements as originators of *tSpec* elements.

[9] A mapping from M can be applied to one or many elements defined in the metamodel *mA*.

In order to ensure semantic equivalence between *Spec* and *tSpec* models, step 1 is broken down into three activities, each discussed in one of the following subsections.

3.1.1 Semantic definition and validation of mappings (activity 1).

The goal of this activity is to establish semantic links between elements of the two metamodels. A deep understanding of the metamodels of the source and target models is necessary to write any mapping between a *Spec* and its *tSpec*. The first part of this section describes the rationale of the activity. The second part talks about the format used to define our mappings.

3.1.1.1 Rationale of semantic definition and validation of mappings

If, for example, the language used in the specification model is UML 1.x and the language used for the design model is ROSE-RT (which includes concepts such as capsules, ports and connectors recently incorporated into UML 2.0), then active classes of the *Spec* will likely map onto *capsules* in the *tSpec*, whereas passive classes will map onto *data classes*. Furthermore, every association of the *Spec* will also have to be scrutinized. In the context of ROSE-RT, an association between active classes of the *Spec* will involve the introduction of *ports* between the corresponding capsules in the *tSpec*. Conversely, an aggregation between two passive classes in the *Spec* will correspond to a containment relationship between two data classes in the *tSpec*.

Also, mappings may be syntactically correct but semantically erroneous. For example, mapping *classes* of the *Spec* into *notes* in the *tSpec* may be syntactically possible but very unlikely to be semantically felicitous! Quite the contrary, such a mapping would unavoidably result in the absence of key structural elements in the *tSpec*. We must emphasize that we are not proposing a solution for the semantic correctness of the mappings' definitions. We emphasize that this activity relies entirely on expertise about the source and target metamodels, and perhaps agreements among *Spec Designers* about semantic relationships that they are establishing. Consequently we must conclude that the validation of the mappings is subjective in nature, as it depends both on the domain of the *Spec* and on one's idiosyncratic approach to bridging the semantic gap

between *Spec* and *tSpec*. However, one must also understand that it is likely that, with time, the reuse in another domain, of the mappings developed and tested for one domain will increase, thereby reducing the effort involved in this activity. Therefore, over time, it will be less likely that we need to address from scratch the correctness of each mapping.

With respect to the properties of the mappings we define, they are unidirectional (from *Spec* to *tSpec*) and may use one or many elements from the source model in order to create/update an element(s) in the target model. Also, a single element in the source model may be used by more than one mapping. Finally, some mappings may be invoked while executing other mapping, thus establishing a dependency relationship among them. For example, mapping an attribute is only possible if invoked while mapping the classifier that owns the attribute.

3.1.1.2 Mapping definition format

Because QVT is still an ongoing process, we instead use a textual definition for our mappings. We first define the name of the mapping, along with a brief description. Then we define the type of the source and target elements. We continue with a function definition that specifies what to do if the mapping is to be applied. We conclude the mapping definition with a constraint defined using OCL. This is the constraint that must evaluate to true if this mapping is to be applied under the assumption that the correct type of source (and possibly target) element has been identified. Here is an example:

Suppose we want to carry active classes from the source model into the target model. And we want to check that those active classes in the source model are not specializations of passive classes. A possible mapping definition follows:

ActiveClassMapping

Description

Maps an active class from a source model into an active class of the target model.

Context

SWRadio Spec : :Class **source**, ROSE-RT : : Model **target**

Actions

```
//create a corresponding new class in the target model
Class target.c1= new Class( );
Target.c1.isActive = TRUE ;
Target.c1.name = source.name;
```

Constraints

```
P1:  // The source element is of type Class
     source.oclIsTypeOf(Class) and
P2:  // The source class is an active class
      source.isActive and
P3:  // The class is not a specialization of another element
      (source.generalization.parent->isEmpty( ) or
P4:  // The class is an specialization of one or more elements
      (source.generalization.parent->notEmpty( )
     // For all the generalized elements in which the source element is an specialization
      and (source.generalization.parent -> forAll (p |
P5:  // Generalized elements are of type Class
          (p.oclIsTypeOf(Class) and
P6:  // Generalized elements are active classes
            (p.isActive))))))
```

In the next section we consider how to verify that the application of this mapping was done in a correct manner.

3.1.2 Correct application of Mappings (activity 2)

Assuming mappings from a *Spec* to its corresponding *tSpec* are semantically correct, the next step consists in verifying that these mappings are applied correctly. To do so, for each mapping we first create what Binder [6, chapter 6] calls a *testable* combinational model represented in the form of truth tables. Each table is based on the constraints defined in each mapping. The combinational model is associated to a 'formalized' method that extracts a suite of tests that adequately *covers* these conditions. We will compare actual vs. expected results to verify if the mappings have been applied correctly.

The definition of the combinational model is done manually. We provide no algorithm or method to define expected results for each combination of values, nor to identify mutually exclusive combinations. We do provide a prototype tool for automatic application of mappings, in which we have included 'automatic categorization' of source elements in order to verify that all valid combinations (from the combinational model) have been included in a test suite. The categorization is automatically performed by the tool by evaluating the properties of each source element and assigning values of 1 or 0 for

each of the mapping constraints. Following the *ActiveClassMapping* example from the previous section, the tool will assign a value of 0 to constraint P1 when evaluating an interface (recall that constraint P1 defines an element to be of type class, and an interface is not). Using also an interface, the tool will also assign a 0 to constraint P2 because an interface cannot be defined as active. The tool will proceed with the evaluation of the rest of the constraints (P3 through P6 in our example) of the mapping. Based on the values assigned for all the constraints, the element is then allocated as one of the possible valid combinations of the mapping (see Table 7). After all elements have been allocated, a report allows us to verify that our test suite does include at least one source element that fits into each valid combination, and that the mapping was only triggered when it was expected to (as defined in the combinational test model).

Continuing with the mapping definition from section 3.1.1, we now consider how to verify its correct application. For convenience, each constraint predicate was numbered. In our example, having 6 predicates implies a total of 64 possible combinations of true and false values. Not all of these combinations are valid though. Predicate P3 defines the absence of a parent class while predicate P4 negates it: these predicates are mutually exclusive making the combination invalid. Also, the combination of the parent not being a class (P5) and being active (P6) is not a valid combination. Following such a semantic analysis, the list of valid combinations that can be evaluated is reduced to 10. If no further semantic reductions are possible, then the specific combinations of predicate values leading to the mapping firing can be made explicit using, for example, a traditional truth table, such as the one presented in Table 7 below:

Table 7. ActiveClassMapping testable combinational model.

	P1	P2	P3	P4	P5	P6	Result	Comments
1	0	0	0	0	0	0	Not Valid	A class cannot have 0 parents (P3) and 1 parent (P4) at the same time
2	0	0	0	0	0	1	Not Valid	A class cannot have 0 parents (P3) and 1 parent (P4) at the same time
3	0	0	0	0	1	0	Not Valid	A class cannot have 0 parents (P3) and 1 parent (P4) at the same time
4	0	0	0	0	1	1	Not Valid	A class cannot have 0 parents (P3) and 1 parent (P4) at the same time
5	0	0	0	1	0	0	0	Source element not a class (not P1), not active (not P2), with parent(s) (not P3 and P4), parent(s) is/are not classes (not P5) and parent(s) is/are not active (not P6).
6	0	0	0	1	0	1	Not Valid	A parent which is not a class (P5) cannot be active (P6)
7	0	0	0	1	1	0	Not Valid	An element which is not a class cannot have a class as parent (P0) and (P5)
8	0	0	0	1	1	1	Not Valid	An element which is not a class cannot have a class as parent (P0) and (P5)
9	0	0	1	0	0	0	0	Source element is not a class (not P1), not active (not P2), with no parents (P3 and not P4), parent(s) is/are not classes (not P5) and parent(s) is/are not active (not P6).
10	0	0	1	0	0	1	Not Valid	If the class has no parent (P3), no properties for the parent can be true (P5 and P6)

11	0	0	1	0	1	0	Not Valid	If the class has no parent (P3), no properties for the parent can be true (P5 and P6)
12	0	0	1	0	1	1	Not Valid	If the class has no parent (P3), no properties for the parent can be true (P5 and P6)
13	0	0	1	1	0	0	Not Valid	A class cannot have 0 parents (P3) and 1 parent (P4) at the same time
14	0	0	1	1	0	1	Not Valid	A class cannot have 0 parents (P3) and 1 parent (P4) at the same time
15	0	0	1	1	1	0	Not Valid	A class cannot have 0 parents (P3) and 1 parent (P4) at the same time
16	0	0	1	1	1	1	Not Valid	A class cannot have 0 parents (P3) and 1 parent (P4) at the same time
17	0	1	0	0	0	0	Not Valid	A class cannot have 0 parents (P3) and 1 parent (P4) at the same time
18	0	1	0	0	0	1	Not Valid	A class cannot have 0 parents (P3) and 1 parent (P4) at the same time
19	0	1	0	0	1	0	Not Valid	A class cannot have 0 parents (P3) and 1 parent (P4) at the same time
20	0	1	0	0	1	1	Not Valid	A class cannot have 0 parents (P3) and 1 parent (P4) at the same time
21	0	1	0	1	0	0	Not Valid	An element which is not a class cannot be active
22	0	1	0	1	0	1	Not Valid	A parent which is not a class (P5) cannot be active (P6)
23	0	1	0	1	1	0	Not Valid	An element which is not a class cannot be active
24	0	1	0	1	1	1	Not Valid	An element which is not a class cannot be active
25	0	1	1	0	0	0	Not Valid	An element which is not a class cannot be active
26	0	1	1	0	0	1	Not Valid	If the class has no parent (P3), no properties for the parent can be true (P5 and P6)
27	0	1	1	0	1	0	Not Valid	If the class has no parent (P3), no properties for the parent can be true (P5 and P6)
28	0	1	1	0	1	1	Not Valid	If the class has no parent (P3), no properties for the parent can be true (P5 and P6)
29	0	1	1	1	0	0	Not Valid	A class cannot have 0 parents (P3) and 1 parent (P4) at the same time
30	0	1	1	1	0	1	Not Valid	A class cannot have 0 parents (P3) and 1 parent (P4) at the same time
31	0	1	1	1	1	0	Not Valid	A class cannot have 0 parents (P3) and 1 parent (P4) at the same time
32	0	1	1	1	1	1	Not Valid	A class cannot have 0 parents (P3) and 1 parent (P4) at the same time
33	1	0	0	0	0	0	Not Valid	A class cannot have 0 parents (P3) and 1 parent (P4) at the same time
34	1	0	0	0	0	1	Not Valid	A class cannot have 0 parents (P3) and 1 parent (P4) at the same time
35	1	0	0	0	1	0	Not Valid	A class cannot have 0 parents (P3) and 1 parent (P4) at the same time
36	1	0	0	0	1	1	Not Valid	A class cannot have 0 parents (P3) and 1 parent (P4) at the same time
37	1	0	0	1	0	0	0	Source element is a class (P1), not active (not P2), with parent(s) (not P3 and P4), parent(s) is/are not classes (not P5) and parent(s) is/are not active (not P6).
38	1	0	0	1	0	1	Not Valid	A parent which is not a class (P5) cannot be active (P6)
39	1	0	0	1	1	0	0	Source element is a class (P1), not active (not P2), with parent(s) (not P3 and P4), parent(s) is/are classes (P5) and parent(s) is/are not active (not P6).
40	1	0	0	1	1	1	0	Source element is a class (P1), not active (not P2), with parent(s) (not P3 and P4), parent(s) is/are classes (P5) and parent(s) is/are active (P6).
41	1	0	1	0	0	0	0	Source element is a class (P1), not active (not P2), with no parents (P3 and not P4), parent(s) is/are not classes (not P5) and parent(s) is/are not active (not P6).
42	1	0	1	0	0	1	Not Valid	If the class has no parent (P3), no properties for the parent can be true (P5 and P6)
43	1	0	1	0	1	0	Not Valid	If the class has no parent (P3), no properties for the parent can be true (P5 and P6)
44	1	0	1	0	1	1	Not Valid	If the class has no parent (P3), no properties for the parent can be true (P5 and P6)
45	1	0	1	1	0	0	Not Valid	A class cannot have 0 parents (P3) and 1 parent (P4) at the same time
46	1	0	1	1	0	1	Not Valid	A class cannot have 0 parents (P3) and 1 parent (P4) at the same time
47	1	0	1	1	1	0	Not Valid	A class cannot have 0 parents (P3) and 1 parent (P4) at the same time
48	1	0	1	1	1	1	Not Valid	A class cannot have 0 parents (P3) and 1 parent (P4) at the same time
49	1	1	0	0	0	0	Not Valid	A class cannot have 0 parents (P3) and 1 parent (P4) at the same time
50	1	1	0	0	0	1	Not Valid	A class cannot have 0 parents (P3) and 1 parent (P4) at the same time
51	1	1	0	0	1	0	Not Valid	A class cannot have 0 parents (P3) and 1 parent (P4) at the same time
52	1	1	0	0	1	1	Not Valid	A class cannot have 0 parents (P3) and 1 parent (P4) at the same time
53	1	1	0	1	0	0	0	Source element is a class (P1), active (P2), with parent(s) (not P3 and P4), parent(s) is/are not classes (not P5) and parent(s) is/are not active (not P6).
54	1	1	0	1	0	1	Not Valid	A parent which is not a class (P5) cannot be active (P6)
55	1	1	0	1	1	0	0	Source element is a class (P1), active (P2), with parent(s) (not P3 and P4), parent(s) is/are classes (P5) and parent(s) is/are not active (not P6).
56	1	1	0	1	1	1	1	Source element is a class (P1), active (P2), with parent(s) (not P3 and P4), parent(s) is/are classes (P5) and parent(s) is/are active (P6).
57	1	1	1	0	0	0	1	Source element is a class (P1), active (P2), with no parents (P3 and not P4), parent(s) is/are not classes (not P5) and parent(s) is/are not active (not P6).
58	1	1	1	0	0	1	Not Valid	If the class has no parent (P3), no properties for the parent can be true (P5 and P6)
59	1	1	1	0	1	0	Not Valid	If the class has no parent (P3), no properties for the parent can be true (P5 and P6)
60	1	1	1	0	1	1	Not Valid	If the class has no parent (P3), no properties for the parent can be true (P5 and P6)
61	1	1	1	1	0	0	Not Valid	A class cannot have 0 parents (P3) and 1 parent (P4) at the same time
62	1	1	1	1	0	1	Not Valid	A class cannot have 0 parents (P3) and 1 parent (P4) at the same time
63	1	1	1	1	1	0	Not Valid	A class cannot have 0 parents (P3) and 1 parent (P4) at the same time
64	1	1	1	1	1	1	Not Valid	A class cannot have 0 parents (P3) and 1 parent (P4) at the same time

Table 8 presents a reduced version of Table 7 with only valid cases from the combinational model shown in the table.

Table 8. ActiveClassMapping testable combinational model (valid cases only).

	P1	P2	P3	P4	P5	P6	Result	Comments
1	0	0	0	1	0	0	0	Source element is not a class (not P1), not active (not P2), with parent(s) (not P3 and P4), parent(s) is/are not classes (not P5) and parent(s) is/are not active (not P6).
2	0	0	1	0	0	0	0	Source element is not a class (not P1), not active (not P2), with no parents (P3 and not P4), parent(s) is/are not classes (not P5) and parent(s) is/are not active (not P6).
3	1	0	0	1	0	0	0	Source element is a class (P1), not active (not P2), with parent(s) (not P3 and P4), parent(s) is/are not classes (not P5) and parent(s) is/are not active (not P6).
4	1	0	0	1	1	0	0	Source element is a class (P1), not active (not P2), with parent(s) (not P3 and P4), parent(s) is/are classes (P5) and parent(s) is/are not active (not P6).
5	1	0	0	1	1	1	0	Source element is a class (P1), not active (not P2), with parent(s) (not P3 and P4), parent(s) is/are classes (P5) and parent(s) is/are active (P6).
6	1	0	1	0	0	0	0	Source element is a class (P1), not active (not P2), with no parents (P3 and not P4), parent(s) is/are not classes (not P5) and parent(s) is/are not active (not P6).
7	1	1	0	1	0	0	0	Source element is a class (P1), active (P2), with parent(s) (not P3 and P4), parent(s) is/are not classes (not P5) and parent(s) is/are not active (not P6).
8	1	1	0	1	1	0	0	Source element is a class (P1), active (P2), with parent(s) (not P3 and P4), parent(s) is/are classes (P5) and parent(s) is/are not active (not P6).
9	1	1	0	1	1	1	1	Source element is a class (P1), active (P2), with parent(s) (not P3 and P4), parent(s) is/are classes (P5) and parent(s) is/are active (P6).
10	1	1	1	0	0	0	1	Source element is a class (P1), active (P2), with no parents (P3 and not P4), parent(s) is/are not classes (not P5) and parent(s) is/are not active (not P6).

The important idea here is, to end up with a *testable* combinational model. Such a model must not only define the conditions under which the mapping fires or not, but also be associated to a 'formal' method that extracts a suite of tests that adequately *covers* these conditions. (An in-depth discussion of methods available to derive sufficient tests from testable combinational models can be found in [6, chapter 6]).

We will consider a transformation from a *Spec* to its transformed *tSpec* to correctly apply mappings if the test suite associated with each mapping is successful (i.e., the mapping fires when and only when it is correct to do so).

In practice, our research suggests this is much less work than it appears to be. Consider, for example, the source and target models given below:

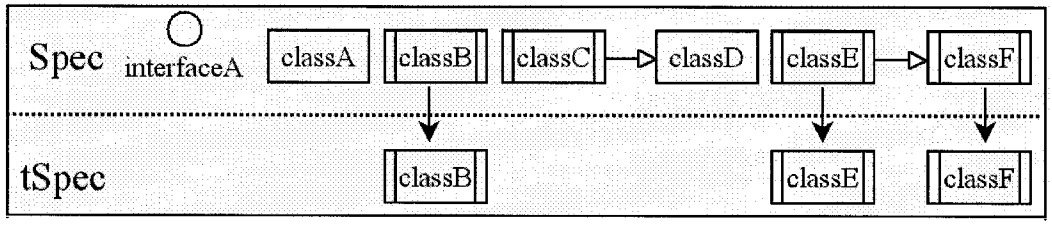


Figure 24. ActiveClassMapping example

We want to verify whether or not the *ActiveClassMapping* from section 3.1.1.2 was correctly applied in Figure 24 in which: 1) Interfaces are illustrated by an empty circle; 2) Passive classes are depicted by a box with the name of the class inside of it, 3) Active classes are illustrated with a box with vertical lines on the sides of the box and 4) generalization relationships are represented by an empty arrowhead as in [76]. Black arrowheads are meant to highlight the application of the mapping.

The element *interfaceA* is not mapped into an active class in the target model because *interfaceA* is not a class (P1). Also, *classA* is not mapped because it is not an active class (P2). Similarly, *classC* is not mapped because it specializes a passive class (P6). Conversely, *classB* and *classF* are mapped into the target model because they are active classes (P1 and P2) with no superclasses (P3). Finally, *classE* is mapped because its combination of properties does satisfy the constraint of the mapping (as captured in the last line of the partial truth table given earlier). Namely: active class (P1 and P2), having a superclass (P4 and thus not P3) and the superclass being an active class (P5 and P6).

The informal analysis from the previous paragraph may be a good start, but we required further proof. We then built (into the tool) traceability relationships between elements of the *Spec* and *tSpec* models. Table 9 presents such traceability when applied to the original model from Figure 24.

Table 9. Traceability relationships *tSpec to Spec* Model Elements

	Spec Model Elements			Mapping	*tSpec* Model Elements			
	Element Type	Name	ID		Element Type	Name	ID	Trace to *Spec*
1	Interface	IntA	S1					
2	Class	ClassA	S2					
3	Class	ClassB	S3	ActiveClassMapping	Class	ClassB	T3	S3
4	Class	ClassC	S4	ActiveClassMapping				
5	Generalization	ClassC-ClassD	S5					
6	Class	ClassD	S6					
7	Class	ClassE	S7	ActiveClassMapping	Class	ClassE	T5	S7
8	Generalization	ClassE-ClassF	S8					
9	Class	ClassF	S9	ActiveClassMapping	Class	ClassF	T7	S9

Table 9 shows us that the *ActiveClassMapping* was intended to be applied on the four active classes from Figure 24, but it succeeded only in three. From this discussion we conclude that, using Figure 24 and Table 9, the *ActiveClassMapping* is tested for the cases where it should apply (i.e., rows 9 and 10 of Table 8) and for some cases where it should not apply. Still we do not have evidence that the mapping has been tested against *all* the cases where it should not be executed (i.e., rows 1 through 8 from Table 8). In order to know which combinations remain to be tested, as explained above, we have implemented into our tool a mechanism that ties each element of a *Spec* model to the combination it uses in a relevant mapping. For example, for the model of Figure 24, we show in Table 10 which elements apply to which combination of the relevant mapping (we have omitted all the non-valid combinations from the table)

Table 10. ActiveClassMapping verification against the testable combinational model (valid cases only)

	P1	P2	P3	P4	P5	P6	Result	Element
1	0	0	0	1	0	0	0	Not in model
2	0	0	1	0	0	0	0	InterfaceA
3	1	0	0	1	0	0	0	Not in model
4	1	0	0	1	1	0	0	Not in model
5	1	0	0	1	1	1	0	Not in model
6	1	0	1	0	0	0	0	ClassA, ClassD
7	1	1	0	1	0	0	0	Not in model
8	1	1	0	1	1	0	0	ClassC
9	1	1	0	1	1	1	1	ClassE
10	1	1	1	0	0	0	1	ClassB, ClassF

From Table 10 we notice that rows 1, 3, 4, 5 and 7 have not been tested. Figure 25 presents a new *Spec* model where *all* possible combinations (i.e., each row of the combinational model) will be covered, thus offering a better test suite for this mapping than the model of Figure 24.

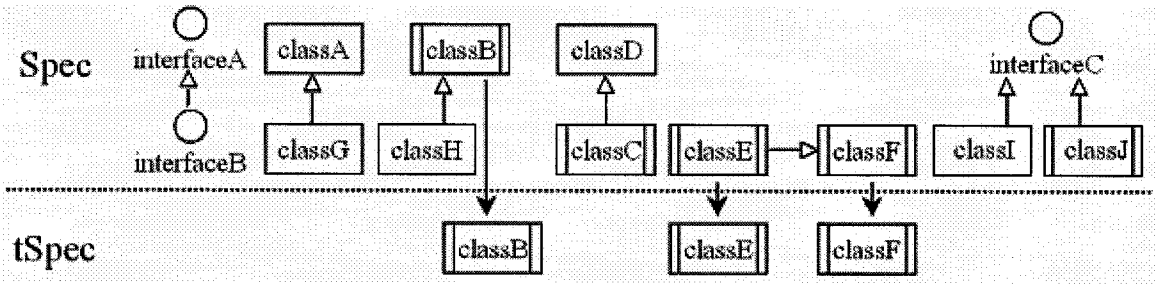


Figure 25. ActiveClassMapping example (revised)

Table 11 presents the tool's output, which demonstrates that our test suite indeed covers at least one case for each of the combinations presented in Table 7.

Table 11. ActiveClassMapping verification against testable combinational model (revised).

	P1	P2	P3	P4	P5	P6	Result	Element
1	0	0	0	1	0	0	0	interfaceB
2	0	0	1	0	0	0	0	interfaceA, interfaceC
3	1	0	0	1	0	0	0	ClassI
4	1	0	0	1	1	0	0	ClassG
5	1	0	0	1	1	1	0	ClassH
6	1	0	1	0	0	0	0	ClassA, ClassD
7	1	1	0	1	0	0	0	ClassJ
8	1	1	0	1	1	0	0	ClassC
9	1	1	0	1	1	1	1	ClassE
10	1	1	1	0	0	0	1	ClassB, ClassF

In addition to the tool support that categorizes each element from the *Spec* model under one valid combination of our combinational model, we have also built in our prototype tool a mechanism that allows us to identify which elements from the *Spec* model have been used as originators of elements of the *tSpec* model. In our internal representation of the *Spec* model, each element contains a property that holds an integer value. Such property starts with a value of 0 for each element, and gets incremented every time that the owner element is used during the execution of a mapping. Following

the example above, after execution of the transformation tool (which at this point considers only the application of the *ActiveClassMapping*) Table 12 is obtained. The table confirms that the mapping was applied only when it was expected to do so.

Table 12. *Spec* elements used while verifying correctness of mapping definitions

#	Element Type	Element Name	Number of times used
1	Class	ClassA	0
2	Class	ClassB	1
3	Class	ClassC	0
4	Class	ClassD	0
5	Class	ClassE	1
6	Class	ClassF	1
7	Class	ClassG	0
8	Class	ClassH	0
9	Class	ClassI	0
10	Class	ClassJ	0
11	Interface	InterfaceA	0
12	Interface	InterfaceB	0
13	Interface	InterfaceC	0
14	Generalization	InterfaceA to InterfaceB	0
15	Generalization	ClassA to ClassG	0
16	Generalization	ClassB to ClassH	0
17	Generalization	ClassD to ClassC	0
18	Generalization	ClassF to ClassE	0
19	Generalization	InterfaceC to ClassI	0
20	Generalization	InterfaceC to ClassJ	0

The table source is a report that logs all the elements from the *Spec* model and the number of times they were used as originators of an element of the *tSpec* model. The same report is also used while verifying the completeness of the mapping definitions that we discuss in the following section.

Our approach for compliance verification considers the application of the described procedure to each and every mapping defined to transform a *Spec* into a semantically equivalent *tSpec*.

3.1.3 Complete definition of Mappings (activity 3)

Validating the semantic correctness of the mappings and then their correct application is not sufficient. We must also consider the completeness of the mappings, that is, we

must verify that all the elements in the *Spec* have been taken into account in the mapping definitions. This step can follow any of the two previous steps. We have found it advantageous to carry steps 2 and 3 together: as we are verifying the correct application of mappings, we can also simultaneously check whether or not any entity that should be mapped is 'forgotten'. Clearly, an incomplete set of mappings will result in having some elements defined in the *Spec* not being transformed into elements of the *tSpec*. We suggest two possible causes for such cases: 1) a mapping is genuinely missing or 2) the constraints of the mapping that should apply are too restrictive.

We return to the example of step 2 to illustrate this. From Figure 24 we identify four different types of elements: an interface (*interfaceA*), passive classes (*classA, classD*), active classes (*classB, classC, classE* and *classF*) and generalization relationships between classes (the white arrowheads). A mapping to map active classes has already been defined. To achieve completeness of the mapping definitions, mappings are also required to map interfaces, passive classes and generalization relationships. In other words, omissions can be easily detected by simply ensuring that all element types of the *Spec* metamodel *do* have corresponding mappings. Such cross-referencing, however, does not deal with the second source of problems.

Finding 'forgotten' elements in small models (such as in the example presented above) may be done visually, but the same approach is difficult to apply in large specifications. To overcome this problem, we build traceability relationships between elements of the source and target models in an automated tool. Roughly put, every element in the source model is assigned a unique identifier that is then 'looked for' in the target model once the transformation is complete. Let us elaborate. The traceability relationship table for our example is presented in Table 13. The first column holds row numbers. The next three columns refer to elements of the specification (*Spec*) model. The fifth column presents mappings that "match" the properties of the source elements, that is, that apply to these elements. Columns six to nine refer to elements in the transformed specification (*tSpec*) model. In columns two and six, Element Type refers to the type of element as defined in the *Spec* and *tSpec* metamodels. Columns three and seven present

the name of the elements as defined in the *Spec* and *tSpec* models. Columns four and eight hold unique identifiers for model elements. Finally, column nine provides traceability information: each element in the *tSpec* model should have at least one element of the *Spec* model as its originator.

Table 13. Traceability relationships *tSpec to Spec* Model Elements

	Spec Model Elements				*tSpec* Model Elements			
	Element Type	Name	ID	Mapping	Element Type	Name	ID	Trace to *Spec*
(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)
1	Interface	IntA	S1	InterfaceMapping	Interface	IntA	T1	S1
2	Class	ClassA	S2	PassiveClassMapping	Class	ClassA	T2	S2
3	Class	ClassB	S3	ActiveClassMapping	Class	ClassB	T3	S3
4	Class	ClassC	S4	ActiveClassMapping				
5	Generalization	ClassC-ClassD	S5	GeneralizationMapping				
6	Class	ClassD	S6	PassiveClassMapping	Class	ClassD	T4	S6
7	Class	ClassE	S7	ActiveClassMapping	Class	ClassE	T5	S7
8	Generalization	ClassE-ClassF	S8	GeneralizationMapping	Generalization	g2	T6	S8
9	Class	ClassF	S9	ActiveClassMapping	Class	ClassF	T7	S9

From Table 13 we notice that *classC* (row 4) from the *Spec* was not mapped into an element of the *tSpec* (neither was row 5 but this is a consequence of not mapping *ClassC* as we explain below). This is not due to the absence of a mapping definition to map active classes, but because of the constraint attached to the ActiveClassMapping definition. In fact, our tool (from which the information on Table 13 was produced) not only tells us which elements were not used as source elements for the *tSpec* model, but also reports the mapping(s) that were attempted to apply on them. The latter provides important clues on the reasons of why the element was not mapped. In this case, the constraint prevents active classes like *classC* being mapped into the *tSpec* because *classC* specializes a passive class (*classD*).

Different causes can be attributed to this problem. For example:

- Cause: The mapping defines incorrectly restrictive constraints. Example solution: Remove from ActiveClassMapping the constraint that prevents an active class that specializes a passive class to be mapped.

- Cause: Mapping constraints are genuinely correct, and there is the need for a new mapping to map the excluded element. Example solution: Define a new mapping to handle the cases excluded by the constraint(s).
- Cause: An omission or error in the structural definition of elements in the *Spec*. The excluded element should possess different properties so that it can be mapped. Example solutions: 1) Redefine *classC* from the *Spec* to be a passive class (or *classD* to be an active class) or, 2) Remove the generalization relationship between *classC* and *classD*.

The actual choice of solution is domain specific. Instead, we want to emphasize here that finding elements of the *Spec* that are not mapped into the *tSpec* may itself lead to the discovery of a deeper problem with the *Spec* specification. Namely, it is possible that the latter contains a genuine semantic error. Should the *Spec* be open to modification, then this error may be corrected. Conversely, if the *Spec* is deemed correct and yet the semantics of the target model require that the banned case be mapped (rather than ignored), then we will have to conclude that the *Spec* cannot be transformed into the *tSpec* using the actual set of mapping definitions.

Aside of the traceability relationships shown in Table 13, we also make use of the counting mechanism described at the end of the previous section. We increase an element's property value by one every time that element is used as originator of an element of the *tSpec*. This time, the aim is to get values of at least one for all the elements in the *Spec* model to verify that indeed the set of mapping definitions is complete. The log report applied to this section's example is shown in Table 14. It shows that two elements from the *Spec* model where not used for mapping application, and thus declared the set of mapping definitions to be incomplete. The reasons were described above.

Table 14. *Spec* elements used while verifying completeness of mapping deffinitions

#	Element Type	Element Name	Number of times used
1	Class	ClassA	1
2	Class	ClassB	1
3	Class	ClassC	0
4	Class	ClassD	1
5	Class	ClassE	1
6	Class	ClassF	1
7	Interface	InterfaceA	1
8	Generalization	ClassD to ClassC	0
9	Generalization	ClassF to ClassE	1

Our approach for compliance verification considers the application of the described procedure until we verify that we have used all elements in the *Spec* model as originators of elements in the *tSpec* model.

3.2 Design to tSpec Compliance

Once the *tSpec* has been verified using step 1 (activities 1, 2 and 3), we assume that, eventually, a *Design* (and more precisely, an executable design) is made available for compliance testing against the *tSpec*. This task is split into two subtasks: first, structural compliance (see section 3.2.1), then behavioral compliance (see section 3.2.2). The key point is that each of these steps is possible first and foremost because the *tSpec* and the *Design* use the *same* underlying metamodel.

3.2.1 Structural Compliance (activity 4)

We perform structural model compliance by analyzing specification and design models, and reporting their similarities and differences. We consider a *Design* being complaint with a specification when the following conditions are met:

1) There exists *Spec*, *tSpec* and *Design* models.
2) The *tSpec* model is semantically equivalent to the *Spec* model as defined in section 3.1.
3) There exist a set of *Compliance Rules* that specifically define the concept of structural compliance.

4) Additional information is provided (when needed) to accommodate for different compliance rules.
5) The *Design* model possesses all the elements, properties and relationships as defined in the *tSpec* model and in accordance with the compliance rules from condition 3.

The aim of finding an exact match between the two models is unrealistic, as many design decisions play a role in defining a design from a specification. When comparing the two models, we consider three different kinds of elements:

1. **'Common' elements** – elements that are defined in the *Spec (and tSpec)* and also in the *Design*
2. **'Only in *Design*' elements** – elements that are defined in the *Design* but are not included in the *Spec* (nor *tSpec*)
3. **'Only in *Spec*' elements** – elements that are defined in the *Spec* (and *tSpec*) but not in the *Design*

Common elements are the simplest kind of elements and are considered a direct match between the specification and the design. The 'only in *Design*' elements are considered additional elements defined in the design to support the specification intended functionality but that are not present in the specification model. We report their presence and do nothing else with them. Thus, in this dissertation, we are left to deal only with "Only in *Spec*" elements.

There are many reasons why an element is in the specification and not in the design. We explore some of them and more importantly, consider the additional information that may lead to an indirect match between elements of both models. We discuss different structural compliance challenges in the following subsections.

In this activity, we first define the properties a *Design* model must exhibit in order to be considered compliant with a specification model. We do so by defining rules for structural compliance. For example, since a *Design* can be thought as a refinement of the

tSpec, it is possible that structural elements will be added, modified and even deleted in it (with respect to the *tSpec*). Having a structural element of the *tSpec* disappear in the *Design* is not necessarily erroneous: in the context of incremental development, at a specific point in time we envision the *Design* may be checked only against a subset of the *tSpec*. But to do this requires that the *tSpec* have its structural elements be annotated with indications such as 'required' versus 'optional'. From the example, we may define two rules that can be evaluated between the models: the first will look for *all* the elements in the *tSpec* to be present in the *Design*, while the second will allow some of them to be absent if declared as optional elements in a separate annotation model. The introduction of compliance rules allows for flexibility in the process, while also focusing on the criteria required to evaluate structural compliance.

We root our structural compliance rules in the nature and origin of the gap that may exist between specification and design models as described in section 2.1. We analyze each aspect and propose a solution. This activity has also been automated using a separate tool (distinct from the one of step 1) that *automatically* verifies structural compliance. This is done by first determining structural differences between the *tSpec* and the *Design* and then verifying if such differences are allowed or not. The discussion of our proposed solutions for structural compliance includes a brief description of how the tool handles each aspect.

We propose two types of rules for structural compliance. The first type considers a policy definition with a model-wide scope. In other words, policies apply to all elements in the *tSpec* and/or *Design* models that fit the description of the rule. For example, addressing the *Package Contents* challenge described in section 2.1.9, the option of enforcing the same package structure (and their contents) of the *tSpec* model to the *Design* model, should be considered a policy rule. In other words, if it is decided that the *Design* is not obligated to follow the same package structure that the *tSpec* model, the policy will apply to all packages in the *Design* model. The second type of rules for structural compliance only impact target elements within the models. For example, addressing the *Naming and Role Playing* challenge described in section 2.1.7, the option

of allowing an element of a design to play the role of one or more elements from the *tSpec* is an element-scope structural compliance rule. The rule will apply only to *the* element in the *Design* that has been defined as implementing one or more elements from the *tSpec.*

We emphasize that although structural compliance tasks (activity 4 in our two-step approach for compliance verification) are carried on between the *Design* and *tSpec* models, some of the structural compliance rules we discuss in this section refer to previous activities that involve transformation/compliance tasks between *Spec* and *tSpec* models. We do so because they address an important aspect of the gap between specification and design models, and because they have direct impact in the creation of the *tSpec* model that will eventually be used for compliance verification of *Design* models.

Table 15 presents a catalog of our structural compliance rules. The first column identifies a sequential rule number. The second column presents the name of the rule. The third column shows the scope (model or element) of the rule. Finally, the fourth column describes the location within the two-step approach for compliance verification that the structural compliance rule is applied.

Table 15. Structural compliance rules

#	Structural Compliance Rule	Scope	Two-step approach activity
1	Representation Language	Model	Transformation *Spec* to *tSpec*
2	Interface Realization	Model	Transformation Spec to *tSpec* Compliance Design to *tSpec*
3	Procedure-call-based vs. signal-based communication paradigm	Model	Transformation *Spec* to *tSpec* Compliance Design to *tSpec*
4	Type correspondence	Element	Transformation *Spec* to *tSpec*
5	Naming and Role Playing	Element	Compliance Design to *tSpec*
6	Mandatory vs. Optional elements	Element	Transformation *Spec* to *tSpec* Compliance Design to *tSpec*
7	Designs implementing only a subset of a specification	Element	Transformation *Spec* to *tSpec* Compliance Design to *tSpec*
8	Package Contents	Model	Compliance Design to *tSpec*
9	Multi-role players	Element	Compliance Design to *tSpec*
10	Specialization	Model	Compliance Design to *tSpec*
11	Design implementing more than one specification	Model	Compliance Design to *tSpec*

Our structural compliance rules are based on additional information that is used to 'annotate' [45] or 'mark' [67] the *tSpec* or *Design* models. Such information is not found in the *tSpec* or *Design* models, and needs to be externally provided. We do so by using XML documents. We further elaborate on this while addressing each of the structural compliance rules.

The remainder of this section discusses our structural compliance rules.

3.2.1.1 Representation Language

This rule considers the possibility that specification and design models use different modeling languages or metamodels. Although our proposed solution for this challenge has been thoroughly explained before (see section 3.1), we remind the reader that structural compliance of a *Design* model is carried against a *tSpec* model and not directly against the specification model. The way mappings are defined, how the transformation from *Spec* to *tSpec* is done and how verification activities from *tSpec* to *Spec* are executed have a direct impact on how we carry on with structural compliance. Should another method not use a *tSpec*, then we believe the whole issue of how to carry out structural compliance would have to be rethought.

We consider the *Representation Language rule* to be within the *General* group of compliance rules. Having a specification model realized using a different language in a design model is a common situation. If we were to pursue the solution of this challenge following the mechanism we suggest in our case study, then we would consider the rule under the *Case Study* group of compliance rules.

3.2.1.2 Interface Realization

The *Interface Realization* challenge arises because there are different approaches to interface realization when using ports to model communication between active classes. Our approach requires a policy for interface realization. The policy defines the guidelines for modeling interface realization using ports. Ports in ROSE-RT, as well as, in UML 2.0 are defined by the set of services that a classifier can provide or request through them.

The concept of port compatibility requires that every service requested by a port must be provided by a connected port, thus establishing a tightly coupled relationship between the two connected ports.

Our proposed solution to this challenge is addressed during the transformation from *Spec* to *tSpec*. That is, we make use of the Interface Realization policy during the transformation process from the specification to the transformed specification.

Continuing with the example from section 2.1.4, we present Figure 26 where an ATM *Spec* model is shown. We also present in Figure 27 and Figure 28 two alternatives for interface realizations. If we define a policy that requires a one-to-one relationship between interface definitions and port declarations, we would consider the design model in Figure 27 to be compliant whereas the design model in Figure 28 to be considered as non-compliant. Alternatively, our policy could have required that an active class realize all of its interfaces with a single port. In this case, Figure 28 would be a compliant design.

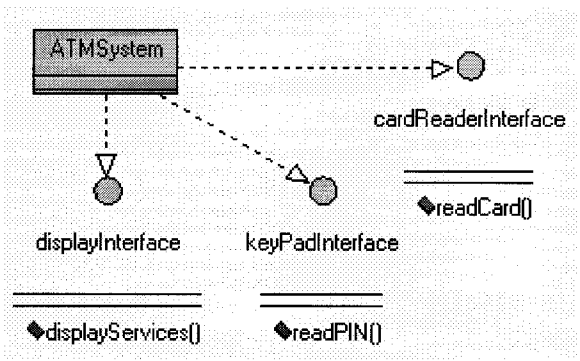


Figure 26. ATMSystem specification model

Although the interface realization challenge is directly addressed by the way mappings are defined and consequently how the *tSpec* model is generated, a policy needs to be available so designers know what would be the requirement with regards to this issue.

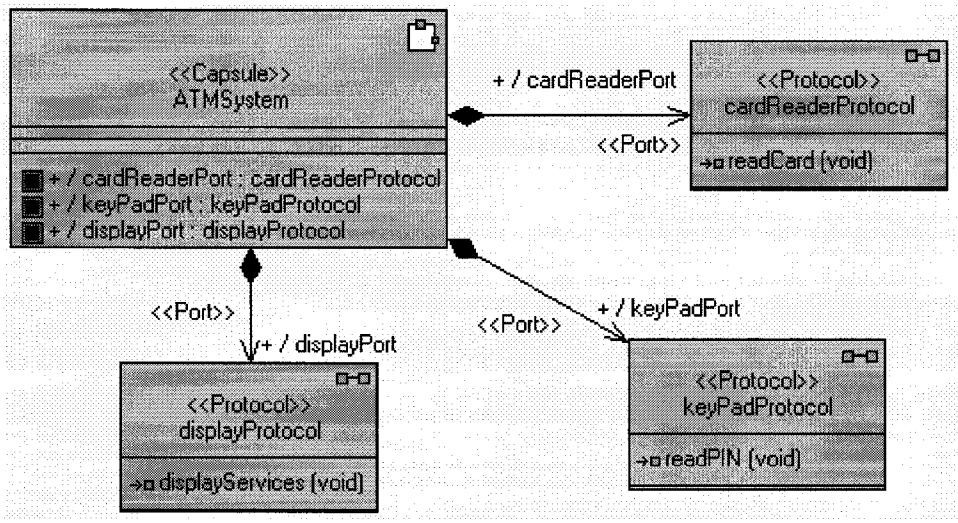


Figure 27. ATMSystem design model with three ports (one port per interface realization)

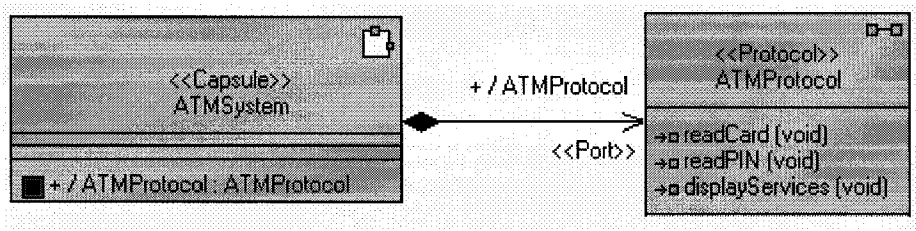


Figure 28. ATMSystem design model with a single port (one port for all interface realizations)

We consider the *Interface Realization* rule to be within the *General* group of compliance rules. In a general sense, the way an interface is realized should be independent of ROSE-RT as well as any information that we have provided within the case study. If we consider that an interface might be realized using protocols, then the compliance rule moves towards the *ROSE-RT* group of compliance rules, as protocols are part of the ROSE-RT metamodel.

3.2.1.3 Procedure-call-based vs. signal-based communication paradigm

In section 2.1.5 we described two major challenges when shifting from a procedure-call-based specification to a signal-based communication design. The first challenge relates to the asynchronous/synchronous nature of the calls, which affects the number of messages that are sent/received for each invocation (signal-based communication use two signals for each invocation defined in the procedure-call approach, while procedure-call-

based communication use one). The second challenge relates to the single piece of data that can be attached to a message in the signal-based approach, as compared to several parameters that can be sent on a procedure call invocation.

We propose to address the first challenge by defining mappings to translate each synchronous procedure-call invocation from the *Spec* model into two asynchronous signals in the *tSpec model.* We refer the reader to section 4.1.1 where the *InteractionMessageMapping* mapping is defined within our case study.

With respect of the data to be attached to a signal, consider the example presented in Figure 29. The service offered by interface1, operation *oper1,* is transformed into two signals in the *tSpec* model: one to offer the service (*oper1*) and a second to return the *int* value (*oper1_RETURN*). We note that the three parameters from *oper1* in the specification model are used to define a new data class that serves as the data type for the service offered (*oper1*) in the *tSpec* model.

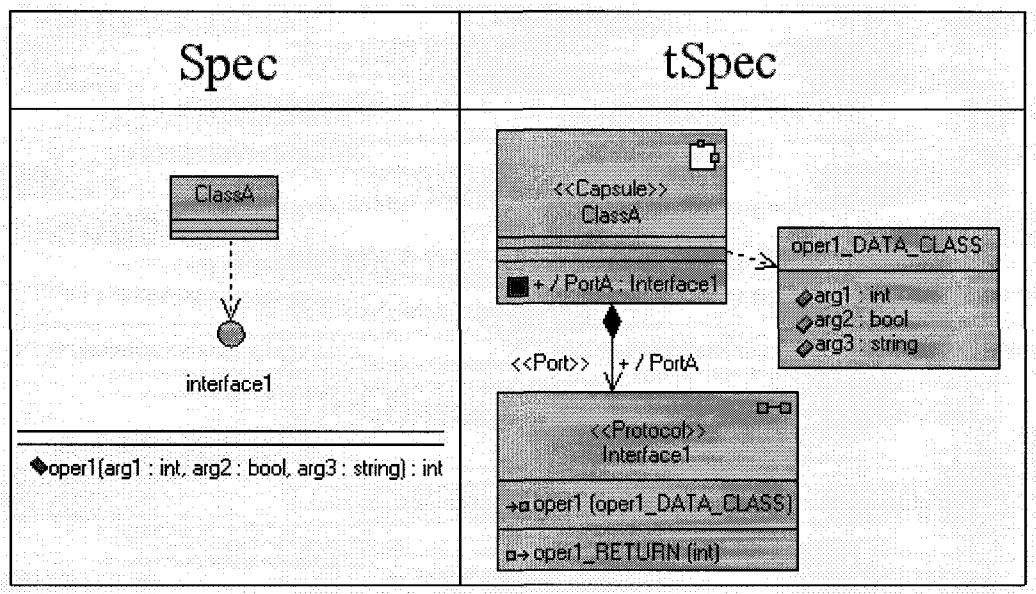


Figure 29. From procedure-call to signal-based communication

The *Design* model then is evaluated for compliance against the *tSpec* as the new representation of the *Spec* model. Our proposed solution also comes in the form of

mappings and we refer the reader to section 4.1.1 where the *OperationToSignalMapping*, *DataClassToSignalTypeMapping*, *OperationResultToSignalMapping*, *OperationToDataClassMapping* and *ParameterToAttributeMapping* mappings are defined. In summary, these mappings and similar ones define how compliance is evaluated. We stress that such mappings apply during the transformation of a *Spec* into a *tSpec* model.

We consider the *Procedure-call-based vs. signal-based communication paradigm* rule to be within the *ROSE-RT* group of compliance rules. The use of signals to achieve communication is coupled with the ROSE-RT tool. Although there are other tools or languages that use a signal-based communication approach, at this point in time we cannot categorize the rule under the *General* compliance rules group, as there are many other cases where procedure-calls are used.

3.2.1.4 Type correspondence

The specification and design models may use different names for abstract types when referring to the same primitive type. We continue with the example from 2.1.6, where some languages may use the type *Boolean* to define a variable that can be either *true* or *false*, while some others may refer to the same primitive type only as *bool*.

Figure 30 presents an example of *Spec* and *Design* models with a type correspondence challenge.

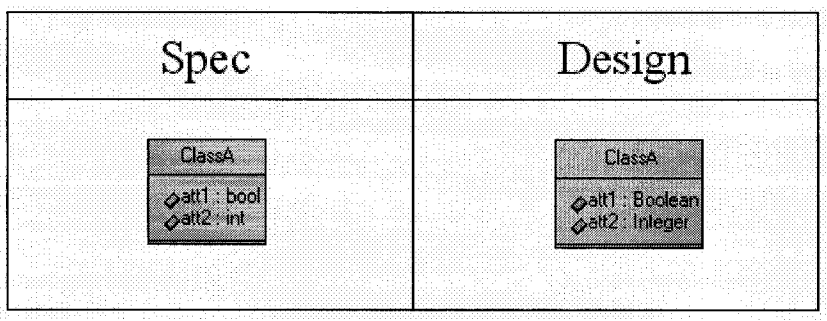


Figure 30. Type correspondence example

For the verification of model compliance, attempting to match *Boolean* attributes from the specification model against *bool* attributes in the design will require additional information. This added information is supplied at the element level and should be reflected in the *tSpec* model. In other words, the *tSpec* model should use *Boolean* and *Integer* data types when referring to *bool* and *int* data types from the *Spec* example in Figure 30. In our prototype tools for model compliance verification, we provide such information using an XML format as shown in Figure 31.

```
<typeCorrespondenceDefinition>
        <typeCorrespondence>
                <sourceType>bool</sourceType>
                <targetType>Boolean</targetType>
        </typeCorrespondence>
        <typeCorrespondence>
                <sourceType>int</sourceType>
                <targetType>Integer</targetType>
        </typeCorrespondence>
</typeCorrespondenceDefinition>
```

Figure 31. Type correspondence information in XML format

The *sourceType* element identifies the primitive data type name used in the specification model while the *targetType* element defines the name used in the *Design* for the same primitive type.

This proposed solution addresses the challenge during the transformation from *Spec* to *tSpec*. The *Design* types are then checked against the *tSpec* types during the verification of *Design* compliance with the *tSpec*. The result of marking the *Spec* model are shown in the *tSpec* model in Figure 32.

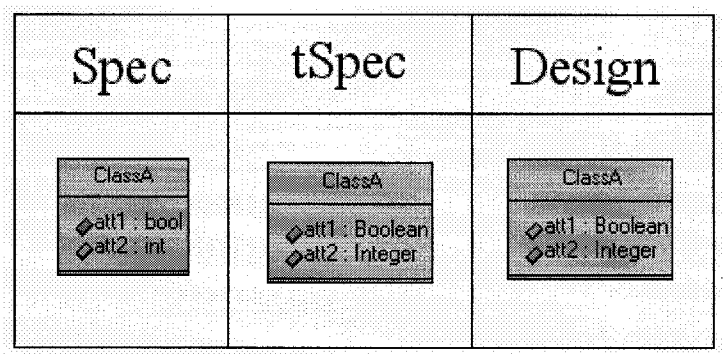


Figure 32. Type correspondence between *tSpec* and Design

Given the additional information of Figure 31, we consider the design from Figure 30 to be compliant with the specification model in the same figure.

We consider the *Type correspondence* rule to be within the *General* group of compliance rules. Having different names in different languages to refer to the same primitive data type is a common situation that is not particular to our case study or ROSE-RT. If we were to pursue the solution of this challenge following the mechanism that we have suggested, then we would consider the rule under the *Case Study* group of compliance rules.

3.2.1.5 Naming and Role Playing

It is possible that an element in the design realizes an element of the specification but uses a different name. Recalling our example from section 2.1.7, Figure 33 presents a class *DomainManager* in the specification model that is realized by the *DM* class in the design. A search for a *DomainManager* in the design will prove unsuccessful because no such element exists (with the same name).

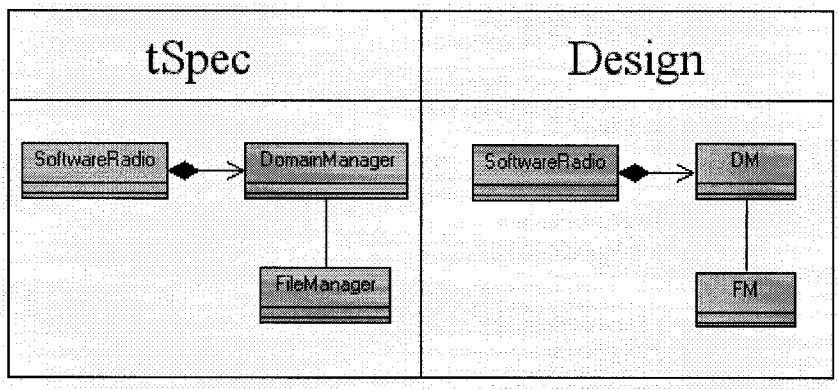


Figure 33. Naming and role-playing example

In order to verify model compliance we propose providing a correspondence table as additional information. The correspondence table matches the names used in the design with the names of elements in the specification. In our prototype tools for verifying model compliance, we provide such information using an XML format shown in Figure

34. Our structural verification tool will look for a *DM* entity in the *Design* model rather than look for the *DomainManager* entity defined in the *tSpec* model.

```
<nameEquivalenceDefinition>
        <nameEquivalence>
                <nameInSpec>Domain Manager</name>
                <nameInDesign>DM</nameInDesign >
        </nameEquivalence>
        <nameEquivalence>
                <nameInSpec>FileManager</name>
                <nameInDesign>FM</ nameInDesign >
        </nameEquivalence>
</nameEquivalenceDefinition>
```

Figure 34. Name correspondence information in XML format

Given such additional information, we consider the design in Figure 33 to be compliant with the specification in the same figure.

We consider the *Naming and Role Playing* rule to be within the *General* group of compliance rules. Having an element in a design model with a name that is different from the one used in the specification is a common situation that is not particular to our case study or ROSE-RT. If we were to pursue the solution of this challenge following the mechanism that we have suggested, then we would consider the rule under the *Case Study* group of compliance rules.

3.2.1.6 Mandatory vs. Optional elements

A specification may state that only a subset of the elements it defines are required (mandatory elements) to be in the design, while the remaining elements are considered optional elements. More specifically, a designer may or may not decide to realize the optional elements. Recalling our example from 2.1.8, Figure 35 presents an example where the specification model defines three elements while a design only realizes two of them. Without any additional information, the *Design* may be deemed non-compliant with the specification.

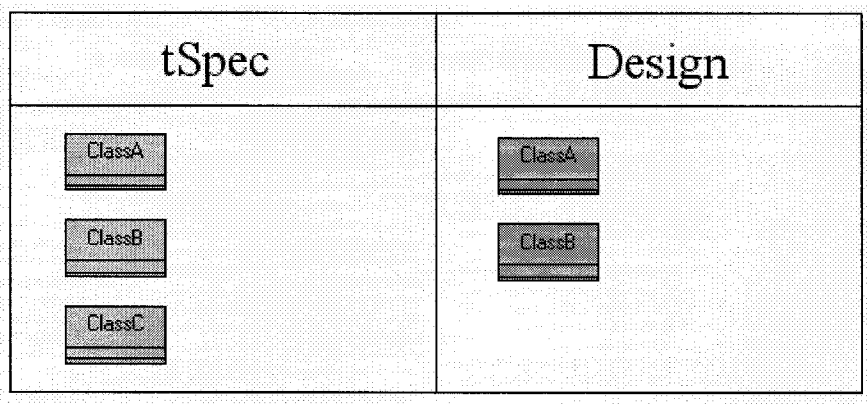


Figure 35. Mandatory vs. optional elements example (unmarked model)

We propose the use of additional information that states the mandatory/optional elements in the *tSpec*. The *Design* is considered to comply with the *tSpec* if it contains all the mandatory elements defined in the latter. In our prototype tools for verifying model compliance, we provide such information using the XML format shown in Figure 36.

```
<optionalElementDefinition>
        <optionalElement>
                <name>ClassC</name>
                <type>Class</type>
        </optionalElement>
</optionalElementDefinition>
```

Figure 36. Optional element information in XML format

The *name* element identifies the name of the optional element in the specification model and the *type* element identifies the type of optional element. Elements not defined as optional are implicitly declared mandatory elements. Providing such additional information is similar to marking the *tSpec* model as shown in Figure 37.

Our proposed solution resolves the challenge in a two-stage process. The additional information is used to create a marked *tSpec*. The markings in the marked *tSpec* are then used during the check for compliance of the *Design*. Provided such additional information, we consider the *Design* in Figure 37 to be compliant with the specification model in the same figure.

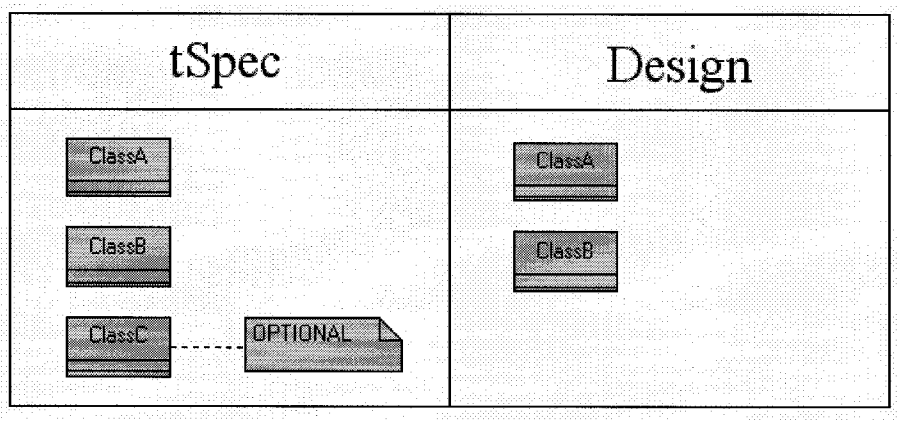


Figure 37. Mandatory vs. optional elements example (marked model)

We consider the *Mandatory vs. Optional elements* rule to be within the *General* group of compliance rules. Having an element in a specification model that is not realized in the design model is a common situation that is not particular to our case study or ROSE-RT. If we were to pursue the solution of this challenge following the mechanism we suggest in our case study, then we would consider the rule under the *Case Study* group of compliance rules.

3.2.1.7 Designs implementing only a subset of a specification

This challenge is similar to the Optional vs. Mandatory challenge from section 2.1.8, the main difference is that the designer has the ability to verify compliance with only a subset of the specification model elements. Even if some elements of the specification are marked as mandatory elements, their absence in the *Design* may not be deemed as non-compliant if such elements are not included in the target subset defined under this condition. Consider Figure 38 where four (4) classes are defined in the specification model and only one of them is considered optional. Allowing the designer to target a subset of the specification, if classes ClassA and ClassB from the design hold the same properties as their counterparts in the specification, then the partial design should be considered compliant, regardless of the missing ClassC (also mandatory) from the specification.

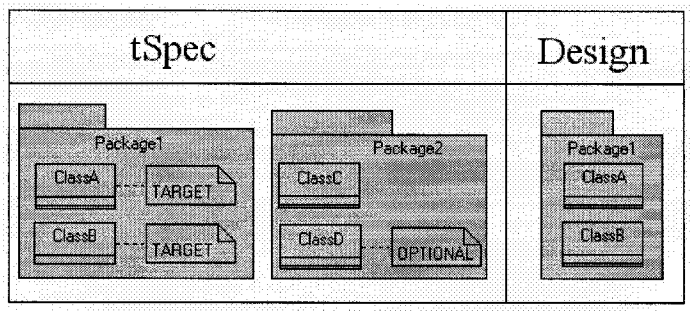


Figure 38. Design implementing only a subset of a specification

Enabling this option allows the designer to check for compliance only the design items that are marked as targets. The implementation of this policy can be package-based (marking Package1 from the design model in Figure 38) thus enabling all the contents of the package to be targeted for compliance verification.

Within our approach to Model Compliance Verification, this proposed solution might be implemented in two ways. The first is to address the challenge during transformation from *Spec* to *tSpec* in which only those model elements tagged as 'target' are transformed into the *tSpec* (and thus the verification of the *Design* against the *tSpec* is not affected). The second is to mark the *tSpec* model and then verify for compliance of the *Design* model only against the *tSpec* elements that have been marked as target. (In this second option the transformation/compliance between *Spec* and *tSpec* is not changed). Although the first solution may be faster in its transformation execution (since it will only create elements in the *tSpec* model for those elements in the *Spec* marked as target), changes on the target element list would require a new execution of the transformation tool. We opted for the second option in which the *tSpec* model is marked (at *Design* to *tSpec* compliance verification time) and the transformation from *Spec* to *tSpec* is executed once.

We consider the *Designs implementing only a subset of a specification* rule to be within the *General* group of compliance rules. Having a design that only implements a subset of the specification is a common situation that is not particular to our case study or ROSE-RT. If we were to pursue the solution of this challenge following the mechanism

we suggest in our case study, then we would consider the rule under the *Case Study* group of compliance rules.

3.2.1.8 Package Contents

The designer may choose a different package structure from that used in the specification. We recall from section 2.1.9 that packages are used to organize software elements. It may be an easy task to verify that a design follows the same package structure as the specification. However, we believe that it is unnecessary for a specification and design to have identical package structures to be compliant. In the end, software components will generally behave in the same way regardless of their container package (with the exception where input/output streams specifically define paths to where the software elements are stored). An example is shown in Figure 39.

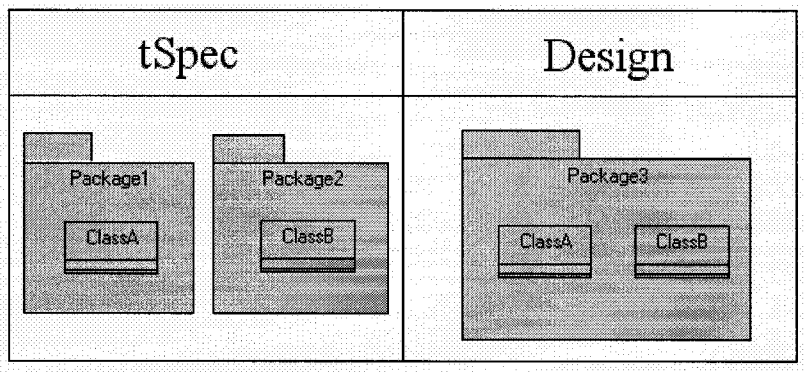


Figure 39. Package contents example

Although we consider the design in Figure 39 to be compliant with the specification in the same figure, we propose that a policy is required stating whether the specification package structure must be followed. The designers then know if they are free to modify the package structure from the specification to one that better fits their needs. This information is used during the verification of compliance between *Design* and *tSpec*.

We consider the *Designs implementing only a subset of a specification* rule to be within the *General* group of compliance rules. Having a design using a different directory structure than the one used in the specification is a common situation that is not particular

to our case study or ROSE-RT. If we were to pursue the solution of this challenge following the mechanism we suggest in our case study, then we would consider the rule under the *Case Study* group of compliance rules.

3.2.1.9 Multi-role players

Allowing a software element in the *Design* to play the role of two or more different entities in the specification may correspond to design decisions and needs to be considered for verifying model compliance. We recall the example we used in section 2.1.10. Figure 40 presents a specification model with two classes and two operations each, while the design model presents a single class that implements four operations. With no additional information, the design should be deemed as non-compliant.

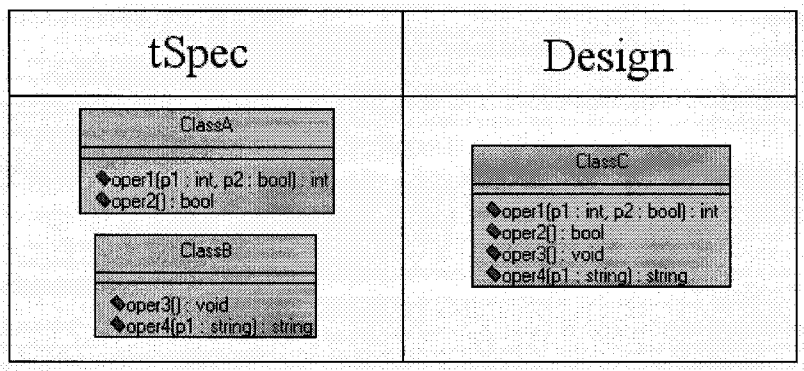


Figure 40. Multi-role player example

Let us consider that the class *ClassC* in the design realizes classes *ClassA* and *ClassB* from the specification. Additional information defining such a relationship allows us to verify that *ClassC* from the design in fact exhibits the structural properties defined for both *ClassA* and *ClassB* in the specification. In our prototype tools for verifying model compliance, we provide such information using XML as in the example below:

```
<nameEquivalenceDefinition>
        <nameEquivalence>
                <nameInSpec>ClassA</name>
                <nameInDesign>ClassC</nameInDesign >
        </nameEquivalence>
        <nameEquivalence>
                <nameInSpec>ClassB</name>
                <nameInDesign>ClassC</ nameInDesign >
        </nameEquivalence>
</nameEquivalenceDefinition>
```

Figure 41. Multi-role player additional information in XML format

Where the *nameInSpec* element identifies the name of the specification element and the tag *nameInDesign* element define the equivalent name used in the *Design*.

Given such additional information, we consider the *Design* in Figure 40 to be compliant with the specification model in the same figure.

We consider the *Multi-role players* rule to be within the *General* group of compliance rules. Having a design using an element that realizes more than one element from the specification is a common situation that is not particular to our case study or ROSE-RT. If we were to pursue the solution of this challenge following the mechanism we suggest in our case study, then we would consider the rule under the *Case Study* group of compliance rules.

3.2.1.10 Specialization

Software elements in the specification may be replaced by specializations of them in the design. We recall the example from section 2.1.11. As Figure 42 shows, the specification model defines associated elements *ClassA* and *ClassB*, whereas in the design, *ClassA* is associated to *ClassC* as a specialization of *ClassB*. Semantically speaking, *ClassC* should possess the same properties, as *ClassB* and thus the association of *ClassA* to *ClassC* in the design should be considered compliant.

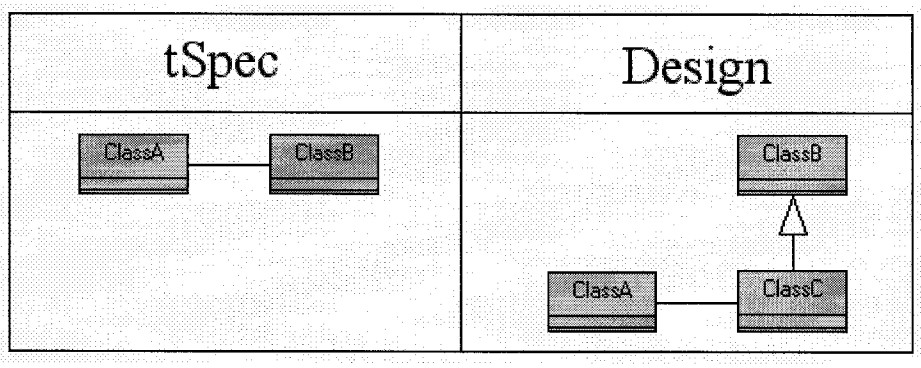


Figure 42. Classifier specialization example

For model compliance verification, this issue needs to be addressed: the process must be able to check for type substitution. That is, the verification process can verify that there is a substitute element, which is a specialization of the original one.

This section refers to classifiers being specialized, but the same rules should apply for properties types and subtypes. In other words, if an attribute of a class is defined as being of *typeA* in the specification model, and that same attribute is found in the design but being of *typeB*, a policy needs to be defined that states that if *typeB,* a specialization of *typeA,* may be substituted or not for compliance purposes. In our work we have included type substitution in our verification tool. First we check for the presence of the specification element in the design model. Then we proceed to verify that the type of both elements is the same. In case the types of the matched elements are different, we evaluate if the design element's type is a specialization or not of the specification element's type. We do so by reviewing the design element's type parents (in case there are parents) and for each parent we check against the specification element's type. We continue to do so until we find a match between the types, or until there are no more parents to evaluate.

In the context of our Model Compliance Verification process, the proposed solution affects the compliance verification step between the *Design* and the *tSpec.*

We consider the *Specialization* rule to be within the *General* group of compliance rules. Having a design using a subclass taking the place of its parent class from the specification is a common situation that is not particular to our case study or ROSE-RT.

If we were to pursue the solution of this challenge following the mechanism we suggest in our case study, then we would consider the rule under the *Case Study* group of compliance rules.

3.2.1.11 Design implementing more than one specification

As explained before, we base structural compliance on comparisons between specification and design models, hoping to find in the design model all the mandatory elements (and their properties) from the specification. If the design contains elements that are not defined in the specification, those elements are reported as being found in the design and not present in the specification, but do not constitute basis to rule non-compliance. In the case of a design implementing more than one specification, most probably all the elements of a design that implements one specification, may be reported as 'only in *Design*' (found in the design but not present in the specification) when checking for compliance against another specification and vice versa. Recall the example presented in section 2.1.2. Figure 43 presents a design model realizing two specifications.

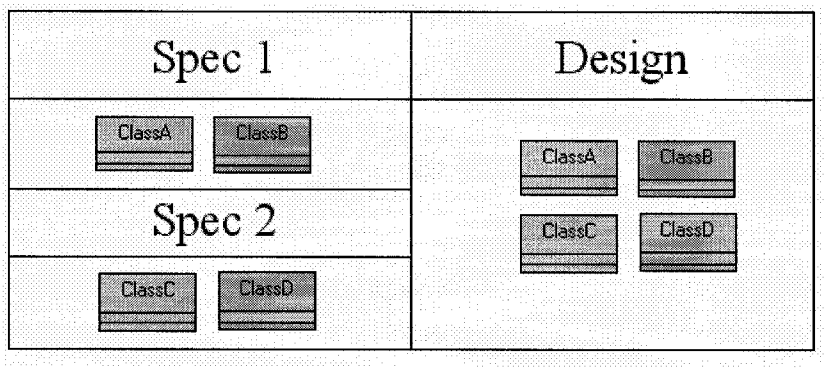


Figure 43. A design implementing two specifications

We consider the *Design* in Figure 43 to be compliant with the specification model in the same figure.

It is important to notice with this challenge that a name duplication of an element in both specifications may lead to an incorrect "matching" of such elements in the design. To avoid such a problem, a structural comparison between the specifications would be required in order to avoid two different elements with different properties to appear in

both specifications under the same name. We have done so using our prototype verification tool. Instead of comparing a *Design* against a *tSpec* model, we use two *tSpec* models that correspond to the two specifications being implemented by a single *Design* model. The tool reports whenever an element is found in both *tSpec* models.

We consider the *Design implementing more than one specification* rule to be within the *General* group of compliance rules. Having a design implementing more than one specification is a common situation that is not particular to our case study or ROSE-RT. If we were to pursue the solution of this challenge following the mechanism we suggest in our case study, then we would consider the rule under the *Case Study* group of compliance rules.

3.2.2 Behavioral Compliance (activity 5)

We define behavioral compliance as the capability of a *Design* model to send/receive a series of message interchanges (between *Design* model elements) as defined in the *Spec/tSpec* model's Sequence Diagrams. This activity requires a design to be defined as a model with execution capabilities. As described before (see section 2.1), ROSE-RT provides the model executability that our approach requires.

We use the compiling and automatic code generation capabilities of ROSE-RT in combination with Rational Quality Architect (RQA) [33] to do this. More precisely, RQA creates test harnesses for our *Design* model, harnesses that enable the automatic generation of run-time sequence diagrams, which are then automatically compared to the specification level sequence diagrams. The results of such comparisons are used to determine the success or failure of the tests. Although this task is a key piece of our overall approach, we must emphasize that we recognize that its implementation may come in many different ways and/or with the support of different tools. In this dissertation we offer one possibility that works in our chosen context (namely MDA and Software Radios). Although we believe that the approach may be applied across different contexts, we do not make such a claim. We also recognize that using sequence diagrams to check for behavioral compliance presents a partial solution to a problem known to be

open. We make no claim or contribution towards solving the problem of checking model execution based on sequence diagrams.

We consider this activity to be tool independent. The following is a list of the properties that such tools must provide:

a) *Model compiling capabilities:* In this first step towards model execution, we use the compiler of the tool to identify any behavioral definition that does not correspond to the structural definition. For example, if in the behavior of the *Design* we define that an instance of *classA* is to be created, and *classA* has not been defined in the structure of the *Design*, then the compiler should catch this error and report it to the designer.
b) *Model execution capabilities*: Our approach to behavioral compliance is dynamic: we require that an execution of the *Design* be 'matched' against the interaction diagrams of the *tSpec*. Thus, the tool should be able to compile a design model into executable code.
c) *Automatic creation of sequence diagrams from an execution of a design model.* This is self-explanatory given that we want to compare the interaction diagrams of the *tSpec* with the ones generated by an execution.
d) *Automatic matching of UML sequence diagrams.* Same explanation as c)

Finally, it is important to remark that given such a tool, the onus is still on the tester to come up with a test suite of executions that adequately 'cover' the possible behaviors of the system (as captured in the *tSpec*). This is not a trivial task, as Binder [6, chapter 6] explains at length.

Testing over several *Designs* (with minor variations between them) allowed us to define a list of possible non-compliance points (i.e., specific locations in the *Design* model that cause non-compliance with its specification model). We used those non-compliant points to provide the designer with a checklist emphasizing where to put special attention when building a design model in ROSE-RT.

We also categorized those non-compliant points according to the aspect (structural/behavioral) they belong to as well as the detection time (compile/execution) at which they can be found. We realize that the detection of some of the non-compliant points to be described in the next subsections could have been incorporated in our transformation tool. We decided against this because our prototype tool is aimed at model transformation and not towards model checking. We also considered that the use of alternate detection mechanisms was enough to prove our contribution without the extra workload involved in embedding our own detection mechanisms in our transformation tool.

Table 16 presents a summary of the causes for non-compliance described in this section. The first column describes the cause for non-compliance. The second identifies if the cause is related to the structural or behavioral definition in the design. The third column presents where the non-compliant point can be detected: at compilation time using ROSE-RT or at execution time using RQA.

Table 16. Causes for non-compliance

Cause for non-compliance	Aspect	Detection time
Required port not defined	Structural	Compile/Execute
Required signal not defined	Structural	Compile/Execute
Required connector not defined	Structural	Execute
Incorrect triggering event	Behavioral	Execute
Incorrect sending of messages	Behavioral	Execute
Missing or Incorrect data in a message	Behavioral	Compile
Messages arriving in a different order	Behavioral	Execute
Inter scenario execution	Behavioral	Execute

In the rest of the section we address separately structural and behavioral non-compliance points.

3.2.2.1 Non-compliance with structural definition

In this section we refer to 'structural readiness' of an entity as a condition in which the entity owns all the structural elements required to perform its intended behavior. We

present examples for the cases where models are not structurally ready. This section is based on the diagrams shown in Figure 44.

Figure 44 a) shows a class diagram with two associated classes (*Class1* and *Class2*). Figure 44 b) presents a behavioral specification using a sequence diagram. *Class1* exhibits three public operations, while *Class2* presents only one public operation. The sequence diagram on the right defines that within the association between *Class1* and *Class2*, *Class2* will call *Class1Oper1* and *Class1Oper3* from *Class1*. *Class1* will invoke *Class2Oper1* from *Class2*.

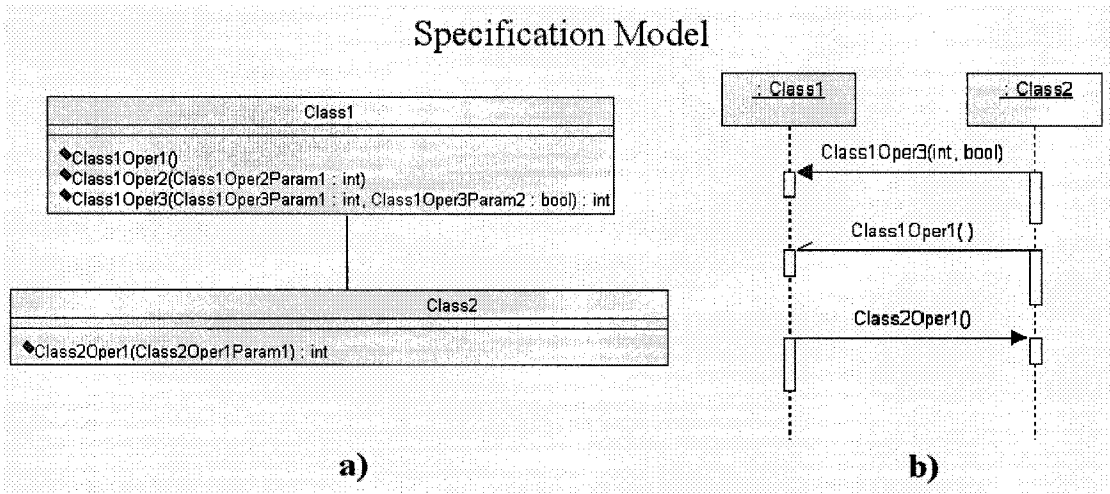


Figure 44. Specification model for compliance verification

As described in section 2.1.5, a change in our communication paradigm requires us to define a communication mechanism between two capsules in order for them to send/receive messages to/from each other. The mechanism includes protocol classes and port definitions. Our intended transformation for the diagrams in Figure 44, results in the diagrams of Figure 45.

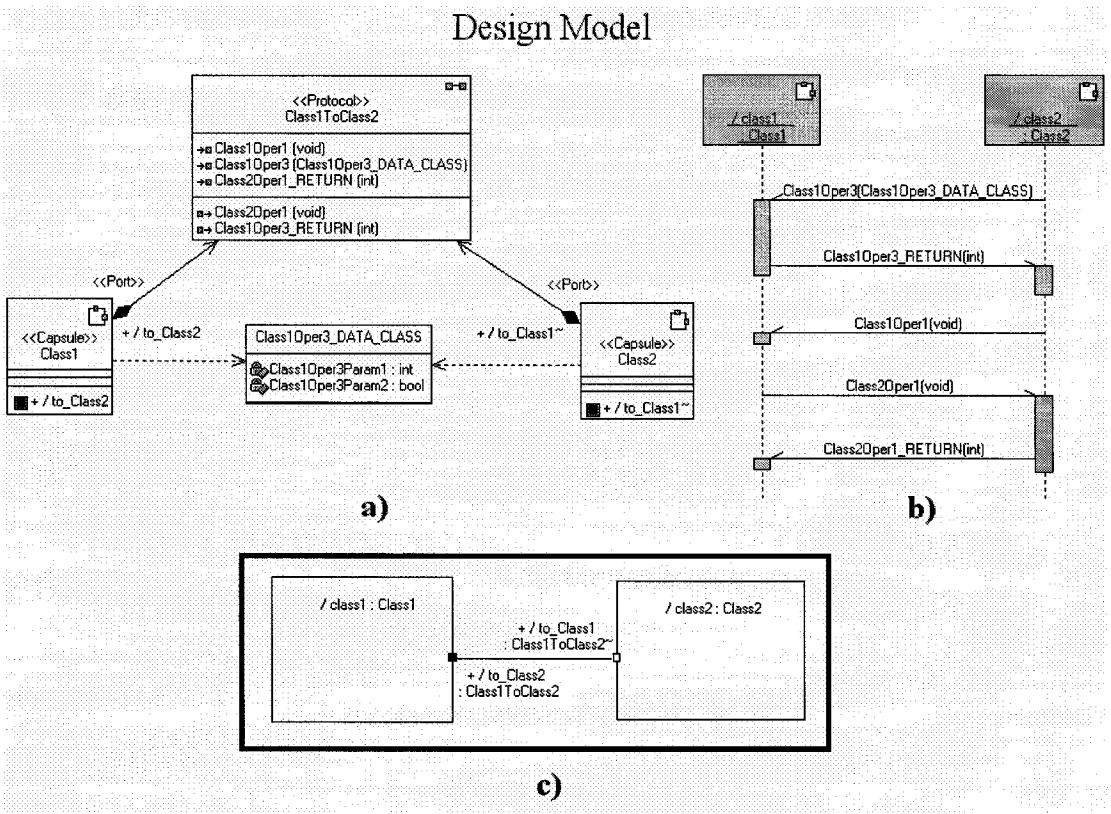


Figure 45. Compliant design model

In Figure 45 a), we present a class diagram with two capsules (*Class1* and *Class2*), each of them with a composition relationship to the *Class1ToClass2* protocol class. Both capsules also contain a dependency relationship that encapsulates the two parameters of *Class1Oper3* from Figure 44. In Figure 45 b), we present an automatically generated sequence diagram that provides two messages (one in each direction) for every synchronous call (that includes a return value) in Figure 44. In Figure 45 c), we present the communication mechanism that includes one public port (shown as a small square box in the boundary of the capsule) for each capsule. The ports are defined to be instances of a protocol class *Class1ToClass2*. A connector links the ports. According to our compliance verification approach, a design model as shown in Figure 45 will structurally comply with the specification model as defined in Figure 44.

We have identified three common causes for failure on structural readiness: 1) a required port has not been defined, 2) a required signal has not been defined, and 3) a

required connector has not been defined. We follow with a detailed discussion for each of the structural non-compliant points.

3.2.2.1.1 Required port not defined

Absence of ports prevents a capsule from communicating with another capsule. Figure 46 presents a situation where *Class1* has no ports to communicate with *Class2*. No communication can be achieved between the two capsules. In Figure 46 a) a composition relationship from *Class1* to the protocol *Class1ToClass2* (in order to define a port) is needed to communicate with *Class2*. A small square box with an X in the middle shows the place of the missing composition relationship. Without a port to communicate with *Class2*, no message can be sent as shown in Figure 46 b). Finally, Figure 46 c) shows the capsule structure diagram with the same square box pointing to the place of the missing port.

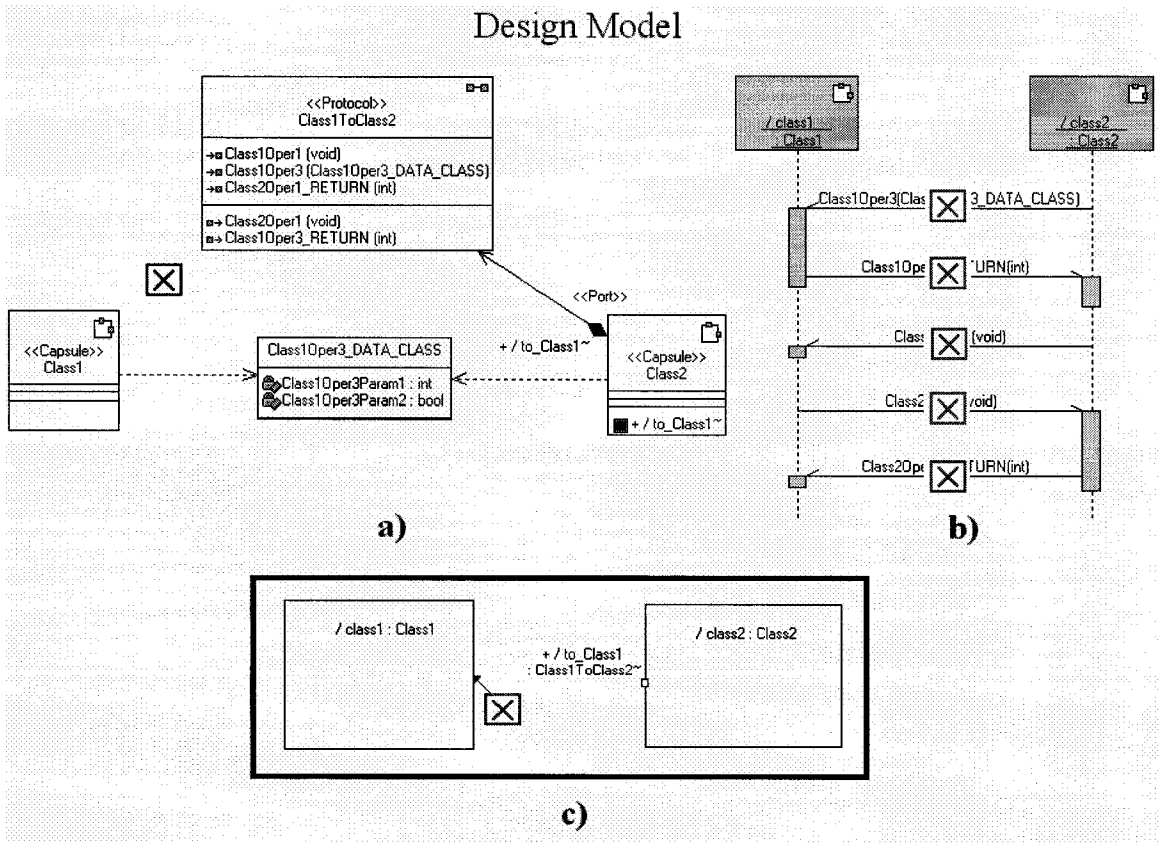


Figure 46. Non compliant design model. Structural readiness failure: Required port not defined

The absence of a required port may be detected at compilation time if a capsule is defined to send a message through a port that does not exist. If no messages are defined to be sent using that port, then the non-compliant point will be detected when RQA attempts to verify the behavior definition while testing the execution of scenarios defined in the design sequence diagrams.

3.2.2.1.2 Required signal not defined

Absence of a required signal also prevents a capsule to successfully comply with the behavior provided by the specification. Figure 47 a) presents a scenario where the communication infrastructure seems to be complete, but the signals *Class1Oper3* and *Class1Oper3_RETURN* are missing (see Figure 45). Without such signals, *Class1* is not able to execute the scenario defined in Figure 44. Even though some communication can be achieved, the correct sequence of messages cannot be obtained.

In Figure 47 a) two signals are missing (indicated by a square box with an X pointing to the place of the missing signals). Signals missing in the *Class1ToClass2* protocol definition prevent *Class2* to send *Class1Oper3* to *Class1* (and also *Class1* to reply to *Class2* with *Class1Oper3_RETURN*) as shown in Figure 47 b).

The absence of a required signal may be detected at compilation time if a capsule is defined to send a message using an undefined signal. If no messages are defined to be sent using that signal, then the non-compliant point will be detected when RQA attempts to verify the behavior definition while testing the execution of scenarios defined in the design sequence diagrams.

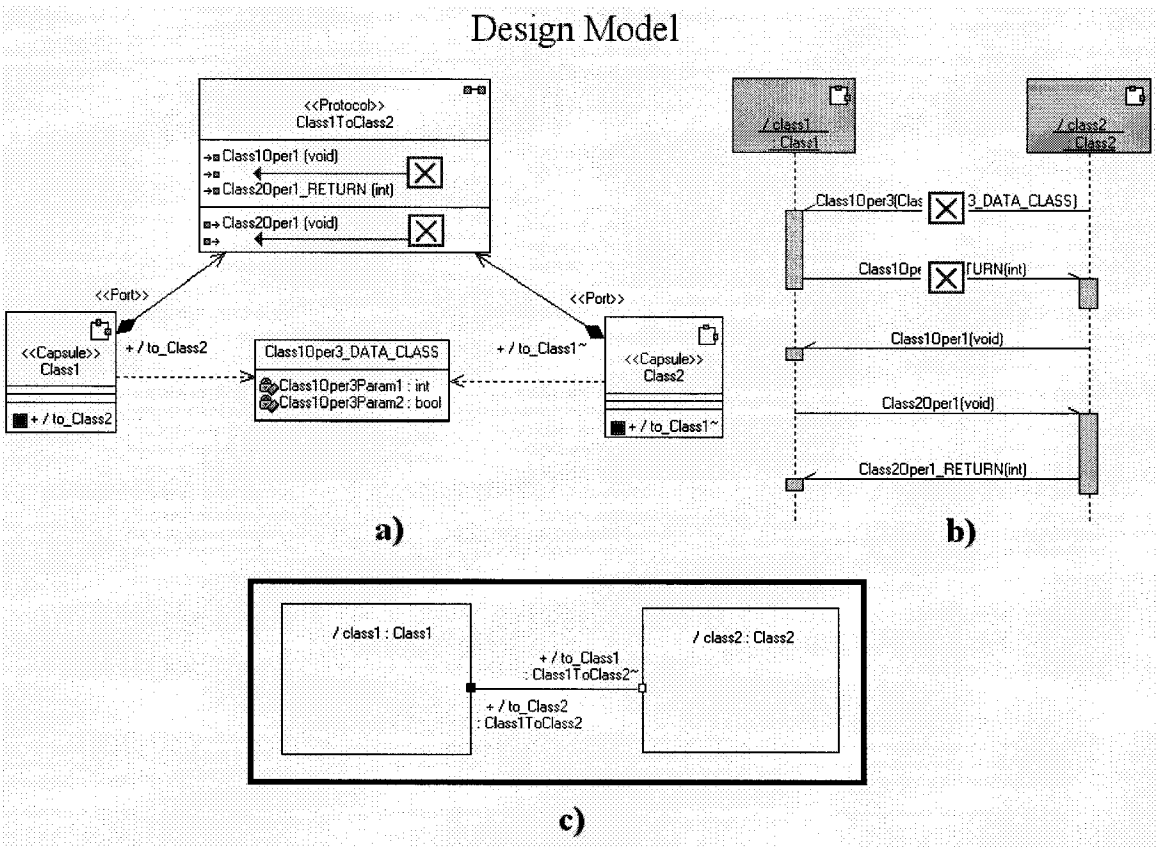


Figure 47. Non compliant design model. Structural readiness failure: Required signal not defined

3.2.2.1.3 Required connector not defined

Absence of a required connector prevents a capsule from successfully communicating with others. Figure 48 presents a scenario where the communication infrastructure is missing a connector between the two capsules. Without such connector, *Class2* is not capable of communicating with *Class1* and thus of executing the scenario defined in Figure 44. No communication can be achieved under these circumstances.

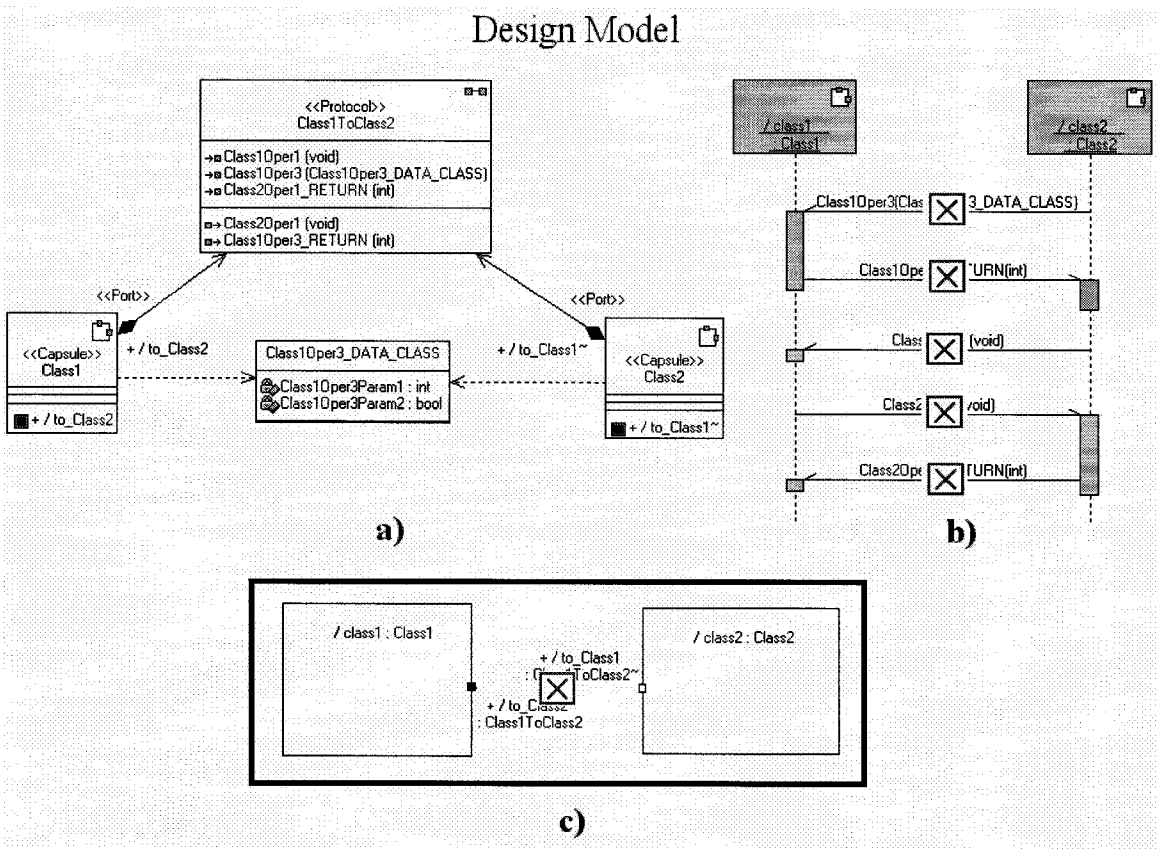


Figure 48. Non compliant design model. Structural readiness failure: Required connector not defined

The absence of a required connector is detected at execution because of the dynamic binding capabilities that ports possess. Connectors bind instances of capsules and are not defined in the capsule definition per se. Instead, connectors are defined on capsule's roles that represent instances of the capsule definition. As it involves capsule instantiation these non-compliant points are detected at execution time.

The remaining of the section explores non-compliant points with the behavioral definition.

3.2.2.2 Non compliance with behavioral definition

In this section, we describe behavioral causes for non-compliance. For this part we make use of RQA. Once again we do not claim to contribute in defining the process to verify the capabilities of an executable design model to comply with a specification

drawn using sequence diagrams. Our contribution comes in the automatic generation of the sequence diagrams to be used in such verification.

The remaining of the section presents a compliant behavioral design and possible causes for non-compliance with the behavioral definition of a design. As we did for the structural causes for non-compliance, in our figures we use small square boxes with an X in the middle to show where the non-compliant point occurs.

In Figure 49 we present a compliant state machine diagram for *Class1* as structurally defined in Figure 45. The diagram presents 3 states (*S1*, *S2* and *S3*) and three transitions (*t1*, *t2* and *t3*). For each transition we provide its triggering event represented by the '*Trigger:*' label followed by the signal name with the data (if any) attached and enclosed in brackets. We also provide in each case the transition code to be executed once the transition has been triggered. The transition code is represented by the '*Code:*' label followed by the port name, signal name, data (if any) enclosed in brackets and the '*.send()*' label

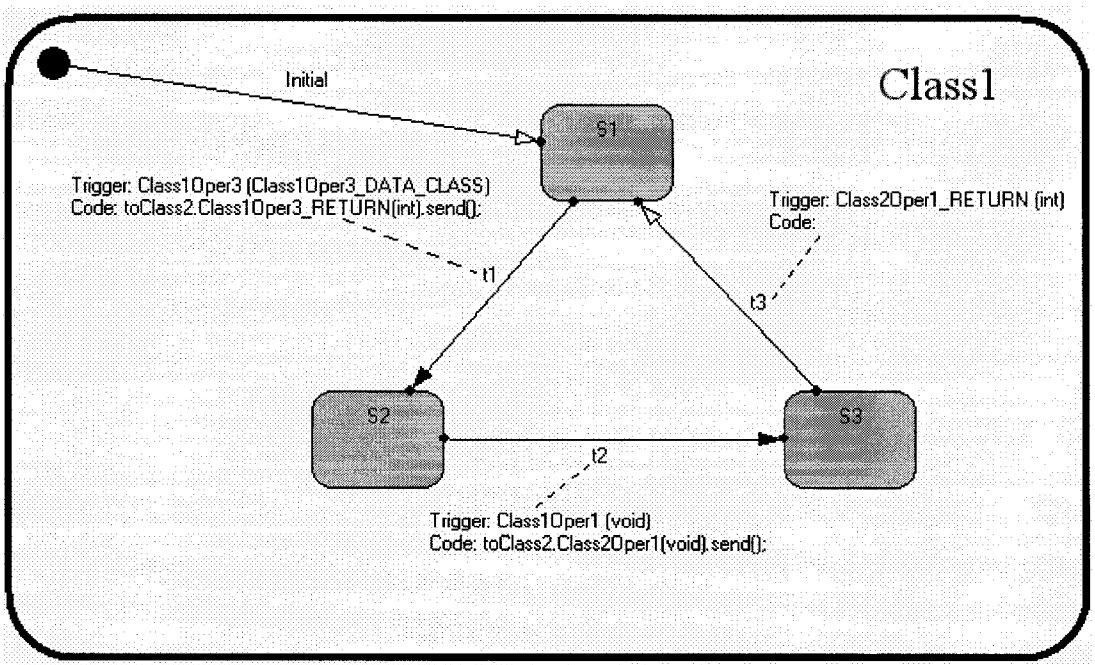


Figure 49. Compliant state machine

The execution of the diagram is as follows: we first go from the *initial state* (represented by a black circle on the top-left corner of the diagram) to state *S1* by taking the initial transition. This transition is taken the first time the state machine is created and requires no triggering events. We simulate the reception of *Class1Oper3* signal from *Class2* with data *Class1Oper3_DATA_CLASS* attached to it. We then execute the code of transition *t1* and send a signal *toClass2.Class1Oper3_RETURN(int).send()* ending up in the *S2* state. We then receive *Class1Oper1* from *Class2* and execute the code in transition *t2 toClass2.Class2Oper1()send()* ending up in state *S3*. We remain in state *S3* until *t3* is triggered. At the reception of the last signal *Class2_Oper1_RETURN(int)* we successfully return to state *S1*. When comparing it to the sequence of events in Figure 45, the flow of messages in the previous exercise will comply with the automatically generated sequence diagram from Figure 45. We consider this design to be compliant with the specification defined in Figure 44.

The next subsections address causes for non-compliance due to incorrect behavioral definition.

3.2.2.2.1 Incorrect triggering event

In this section we present a non-compliant behavioral definition due to incorrect selection of triggering events on a state machine definition. Figure 50 presents a state machine similar to the one of Figure 49. It has also three states (S1, S2 and S3) with three transitions (t1, t2 and t3) defined with their triggering events and code. The main difference lies in values given to the triggering events and code to be executed for each transition.

The execution of the state machine in Figure 50 (assuming the same behavior from Class2) is as follows: after the initial state and initial transition we end up in state S1. First we receive Class1Oper3 signal from Class2. In this case the signal is dropped because S1does not include a transition that is triggered by such signal. Class2 will then wait to receive the return value attached in the signal Class1Oper3_RETURN... value that under the presented circumstances will never arrive thus preventing Class2 from

sending the Class1Oper1 signal. This results in a deadlock situation. The same state machine under option 2 is thus declared in non-compliance with the sequence diagram definition from Figure 45, making the whole design to be not compliant with the specification model defined in Figure 44.

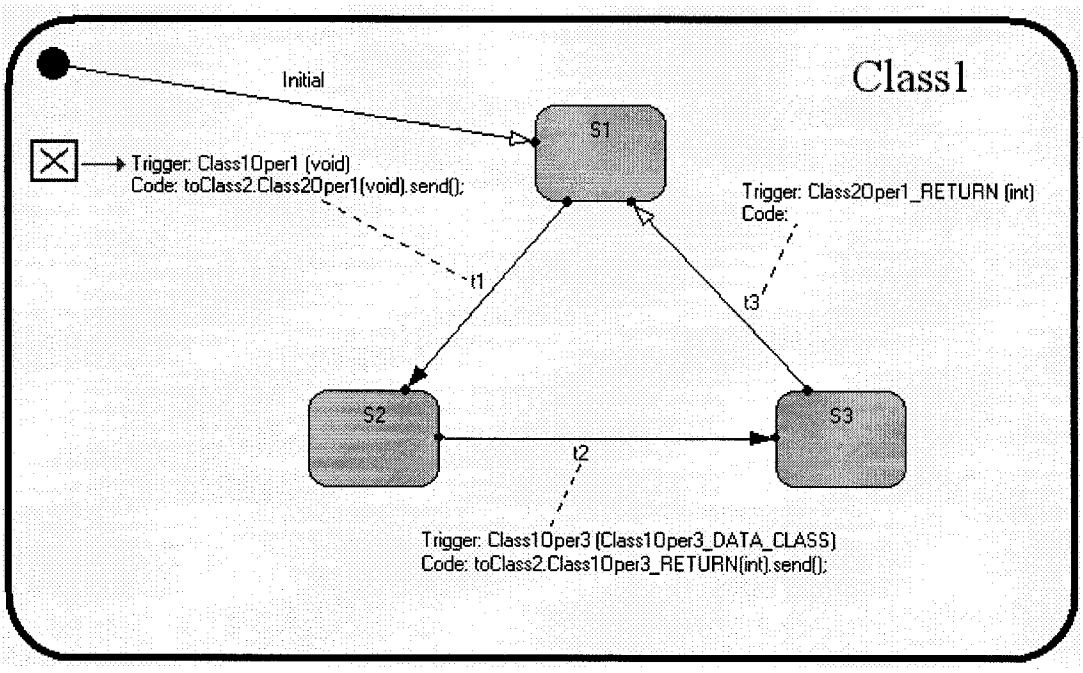


Figure 50. Non-compliant state machine due to incorrect triggering event definition

The incorrect triggering event definition is detected at execution time. At compile time, the compiler only looks for the existence of a signal definition that, upon its reception, will trigger the transition in which it is embedded.

3.2.2.2.2 Incorrect sending of messages

In this section we present a non-compliant behavioral definition due to incorrect sending of a message on a state machine definition. Figure 51 presents a similar state machine as defined in Figure 49. It has also three states (*S1*, *S2* and *S3*) with three transitions (*t1*, *t2* and *t3*) defined with its triggering events and code. The main difference stems from values given to the code to be executed for the first transition.

The execution of the state machine in Figure 51 (assuming the same behavior from *Class2*) is as follows: after the initial state and initial transition we end up in state *S1*. We simulate the reception of *Class1Oper3* signal from *Class2* with data *Class1Oper3_DATA_CLASS* attached to it. We then execute the code of transition *t1* and send a signal *toClass2.Class2Oper1(void).send()* ending up in the *S2* state. At this point Class2 will drop the message because it is expecting to receive the signal *Class1Oper3_RETURN(int).send()*. This situation leads us to a deadlock similar to the one described in the previous section. This behavioral definition is thus declared in non-compliance with the sequence diagram definition from Figure 45, making the whole design to be not compliant with the specification model defined in Figure 44.

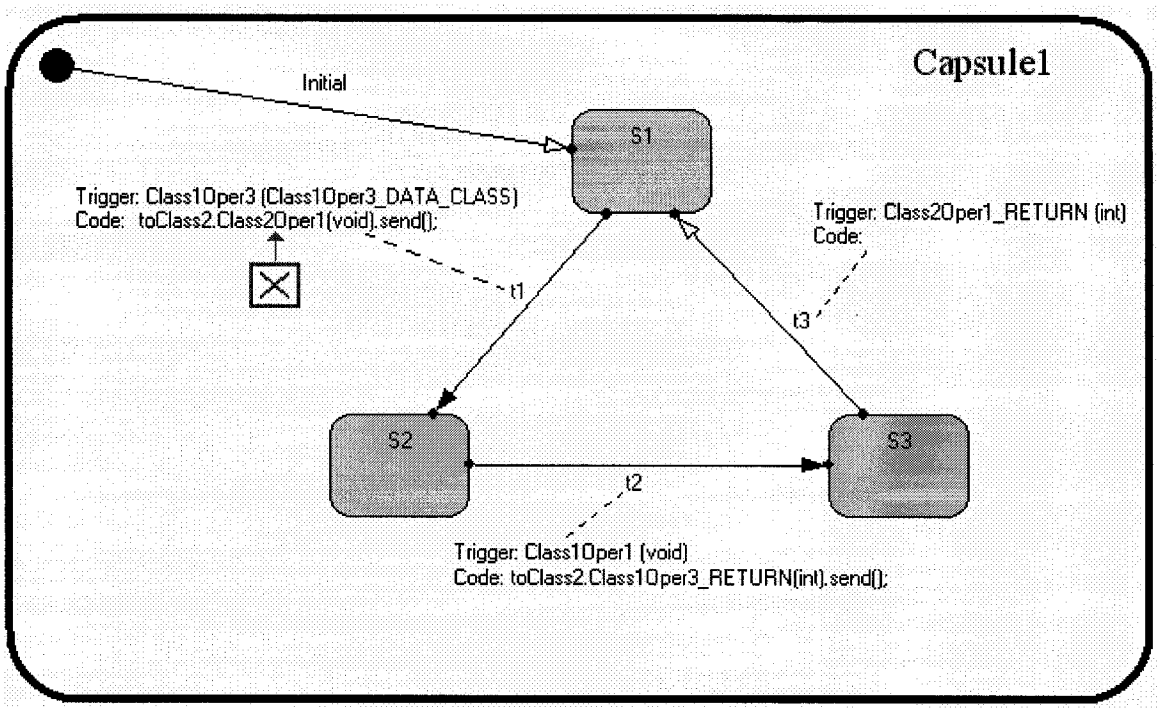


Figure 51. Non-compliant state machine due to incorrect message sending

The incorrect sending of messages is detected at execution time. At compile time, the compiler only looks for the existence of a signal definition that is used to define a message to be sent.

3.2.2.2.3 Missing or Incorrect data attached to a message

In this section we present a non-compliant behavioral definition due to incorrect data attached to the definition of a message to be sent. Figure 52 presents a similar state machine as defined in Figure 49. It has also three states (*S1*, *S2* and *S3*) with three transitions (*t1*, *t2* and *t3*) defined with its triggering events and code. The main difference lies in data attached to the definition of the message to be sent in transition *t1*.

A compile time error results on the non-execution of the state machine defined in Figure 52. The compiler verifies that the data attached to a message to be sent is of the same type of the one used in the signal defined in a protocol class. In this case and as defined in the protocol class from Figure 48, the signal *Class1Oper3* is expected to send an object of type *Class1Oper3_DATA_CLASS*. Instead the definition of transition *t1* on Figure 52 defines and *int* value to be sent attached to the *Class1Oper3* signal. This situation is caught by the compiler and produces a compile error. As mentioned before, any non-compliant cause detected at compile time prevents execution, and thus causes this behavioral definition to be declared in non-compliance with the sequence diagram definition from Figure 45. The whole design is then declared to be not compliant with the specification model defined in Figure 44.

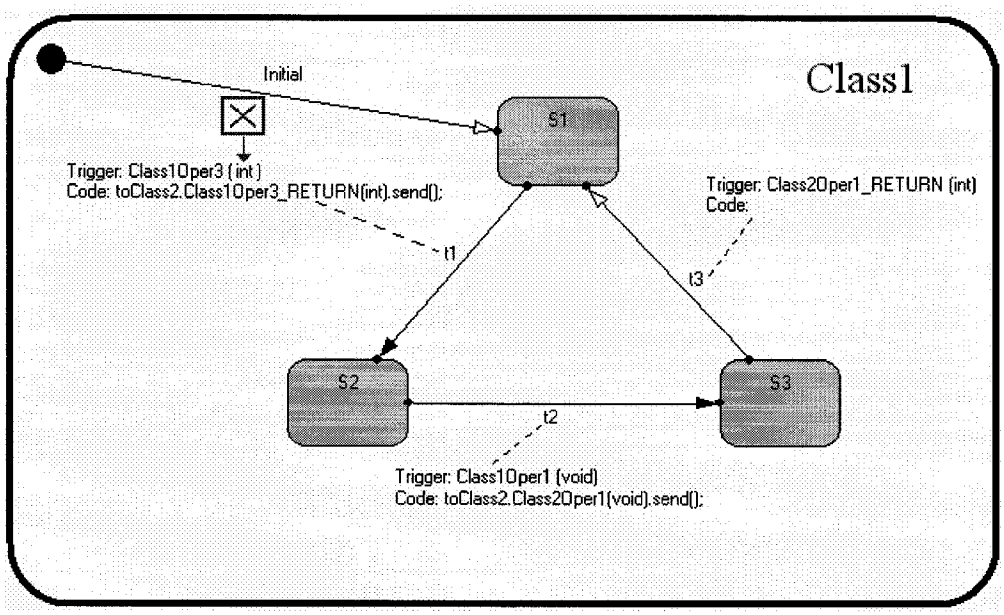


Figure 52. Non-compliant state machine due to incorrect message sending

3.2.2.2.4 Messages arriving in a different order

In this section we present a non-compliant behavioral definition due to messages arriving in a different order than what it was specified. An example is presented in Figure 53 that includes two sequence diagram definitions. They both present the same information but differ in the order in which messages 3 and 4 are sent.

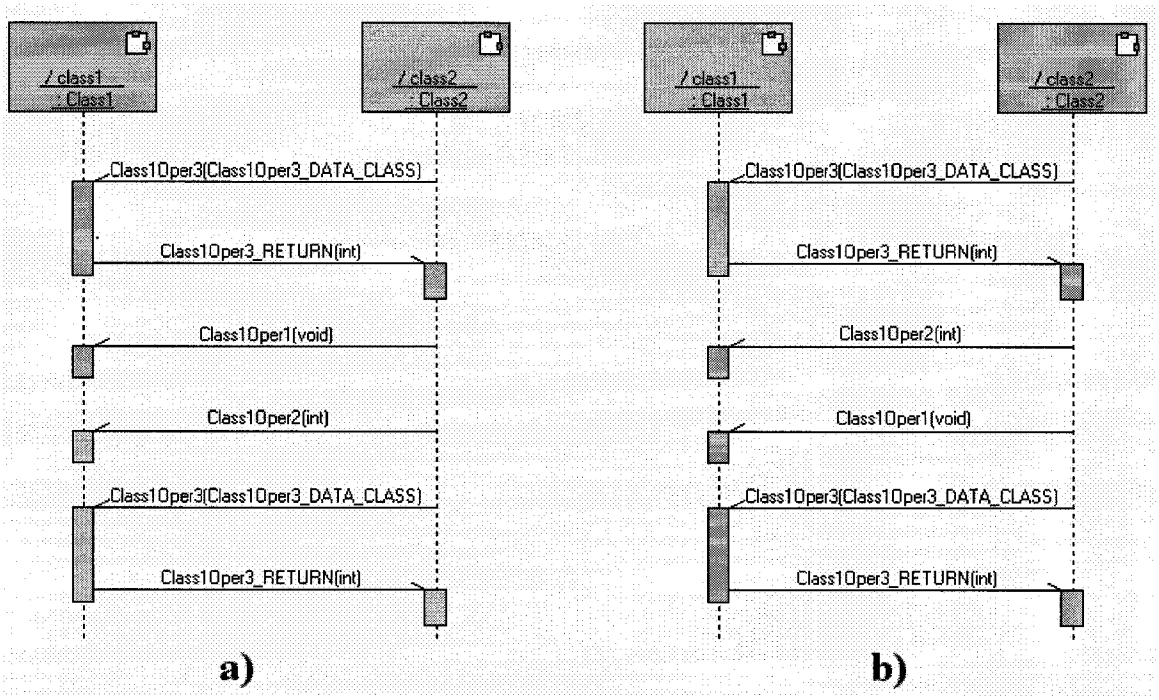


Figure 53. Sequence diagram definition with different message ordering

We use the diagrams from Figure 53 and discuss three different cases that are based on whether the diagrams are defined in the *tSpec* or represent runtime executions of the *Design* model:

1) Figure 53 a) represents a *tSpec* sequence diagram to comply with, while Figure 53 b) represents a runtime execution of an actual message interchange.
2) Both sequence diagrams represent *tSpec* sequence diagrams to comply with, and thus two separate executions will be needed to test for compliance.
3) Both sequence diagrams represent *tSpec* sequence diagrams, but it is only required to comply with one of the two sequence diagrams. In other words, the

order of arrival of messages 3 and 4 is not mandatory as long as they arrive after message 2 and before message 5.

For the first case RQA will output a failed test because the messages arrived with a different order than the one in which they were specified. In this case the assumption is that the order of the messages needs to be strictly enforced. A non-compliant point will be detected and the design will have to be changed so it can accommodate the message interchange in the same order as provided in Figure 53 a).

The second case is similar to first case inasmuch as both are seeking to execute the message interchange in the exact order as defined. In this case two sets of message interchanges will need to be provided from execution to prove compliance with Figure 53. We do not present an additional figure of a compliant set of sequence diagrams because it would be exactly the same as of Figure 53. Any variation would mean non-compliance.

The third case states that it is not important the order in which messages 3 and 4 arrive, as long as they arrive and also that they do it after message 2 and before message 5. For this case there is an additional element in ROSE-RT to denote such situation. Figure 54 presents a new diagram where two thick lines in the lifeline of Class1 enclose messages 3 and 4. The new element defines that the ordering in which those two messages arrive is not relevant for the compliance verification of the sequence diagram. The solution provided for this last case is considered an addition to the automatically generated sequence diagrams of our approach, and represents a semantic variation from the specification level sequence diagrams. In other words, the case study we used does not have its sequence diagrams use this feature. Consequently, no mappings were defined nor implemented in the transformation tool.

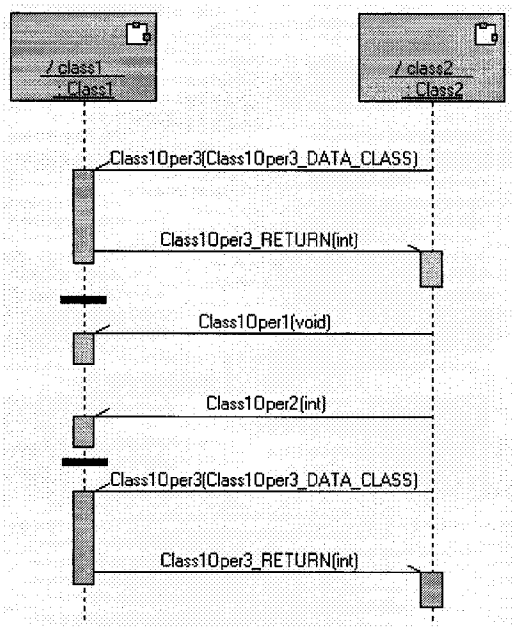


Figure 54. Sequence diagram definition handling messages with different arrival ordering.

With the addition of the latter diagram, if Figure 54 is the sequence diagram to comply with, both diagrams of Figure 53 (if they are the result of execution message interchange) would be considered by RQA as success tests and thus compliant.

The next section presents an overall discussion of the impact of automated support in our approach.

3.3 Automated Support Impact in our Two-step approach for model compliance

In this section we discuss the overall impact of automation within our approach. The goal of this section is to present some specifics about of our prototype tools as well as to provide the reader with discussion about the impact and forces that drove the definition of each part of the tools. The section also highlights what we have learned through them. As outlined in section 1.6, the real contributions of our prototype tools are not the tools themselves, but the principles behind them.

Figure 55 presents an activity diagram similar to Figure 22 in Chapter 3. This time the purpose is to highlight tool support for the main five activities (and their subactivities) of our two step approach for compliance verification. Subactivities supported by our prototype tools are denoted by a white 'T' enclosed in a black filled circle at the bottom right corner of the subactivity. Subactivities supported by Rational tools are denoted by a black 'T' enclosed in a white filled circle at the bottom right corner of the subactivity.

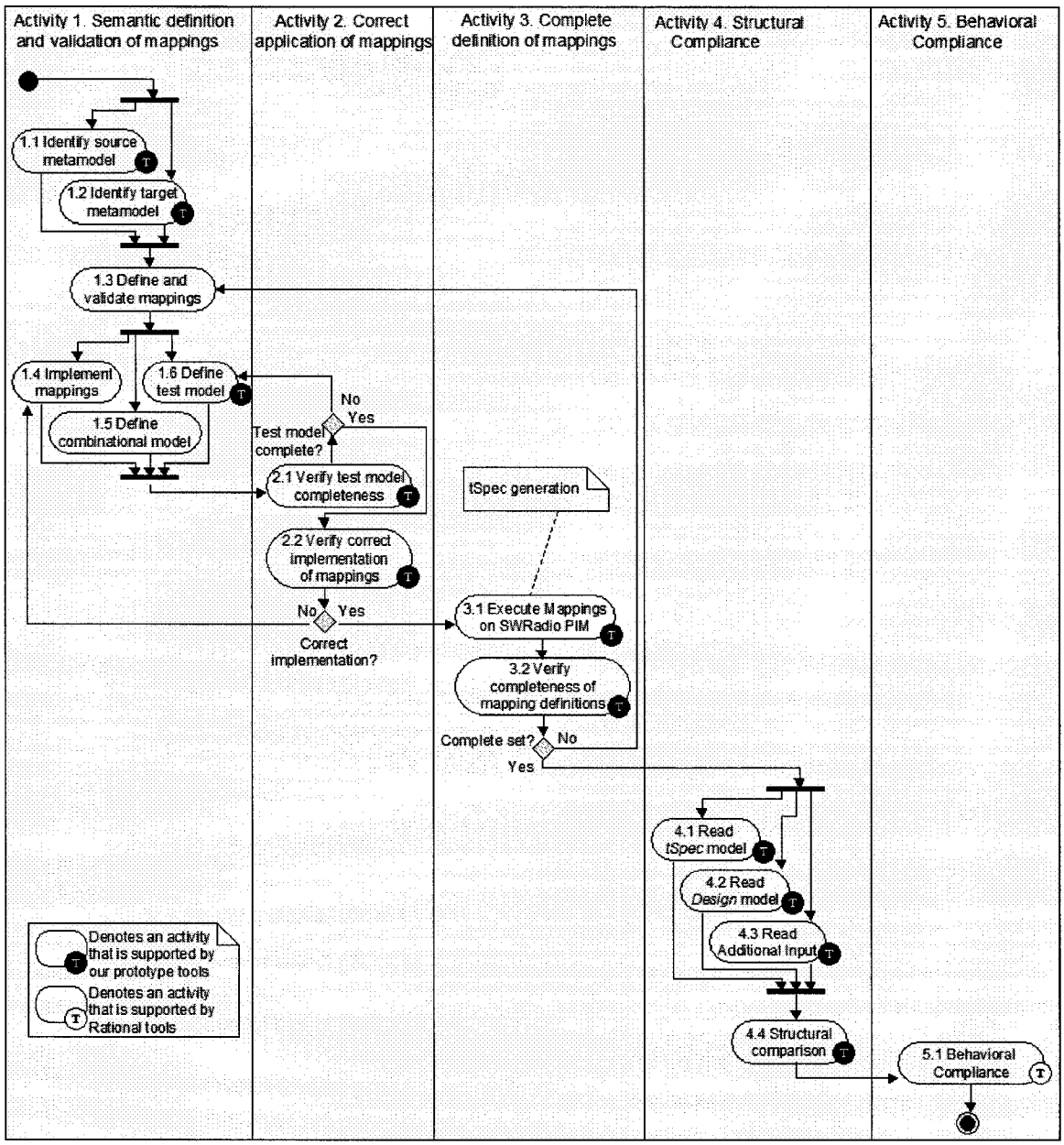


Figure 55. Tool support for the two step approach for compliance verification.

In the remainder of the section we discuss in detail the automated support that tools provide to our approach. We divide the section in four parts. The first three are related to tool support for the *Spec* to *tSpec* transformation and compliance, while the fourth addresses tool support for structural compliance of a *Design* with the *tSpec* model.

3.3.1 *Spec* to *tSpec* Transformation Automation

This section describes the forces behind the building of our prototype tool with respect to automating the transformation process. We start by recalling the high level context of our work: Model Driven Development (MDD), its reliance on automation, and its key premise that programs are automatically generated from their correspondent models [90]. Deeply immersed in MDD, our two-step approach for compliance verification also relies heavily in automation. This section briefly addresses the first force behind our prototype transformation tool: the automatic generation of a semantically equivalent intermediate model (*tSpec*) that uses the same language as the design models we want to verify for compliance (see section 3.1).

For simple models that use perhaps few dozen of elements and relationships, the manual creation of such semantically equivalent intermediate model might be feasible. But we remind the reader that we are working with standard specifications that define several hundreds of elements and relationships (see section 2.3.3). The complexity of our source models makes the manual creation of such intermediate models impractical and not reliable. Model to code transformation has proven feasible and is available in commercial tools (e.g., Rational Rose RealTime © (ROSE-RT) [33], Objecteering Tests for Java/EJB © [61], I-Logix Raphsody © [31] or Telelogic Tau © [100], etc.). With our work we have provided an instance of model-to-model transformation that is: 1) different from model-to-code transformation, 2) executed earlier in the development process and, 3) contributing to the overall MDD goals.

This first stage in the automation support for our approach provided us with valuable knowledge that permeated through the rest of the tool support infrastructure. Activities 1.1 and 1.2 from Figure 55 refer to the identification of source and target metamodels.

Metamodel elements are found in the source model in the form of 'element types' of the source model elements. Activities 1.1 and 1.2 from Figure 55 are about identifying 'element types' from the source and target models. During the identification process, we found 'element types' that were of interest for our compliance purposes (e.g., classifiers, properties, relationships, etc.) and also found others that were not (e.g., graphical shapes, coordinates of elements in diagrams, etc.).

We had a very good idea of the elements and relationships used in the SWRadio PIM but we needed to verify that we were in fact gathering all the important element types for our source/target metamodels. At that point we found the first contribution of automation to our work. In finding the *Spec* metamodel, we visually went through our *Spec* model and gathered most of the element types used in the model. But manual inspection was no match against automatic support. We defined a mechanism that, while parsing the source model, was also producing output into a log file. The log file, that we called 'InfoNotConsidered.txt', stored which lines from the source model were being used towards identification of "element types" as well as which lines were not. Lines used for mapping were then replaced by blank lines into the log file, while lines not used were identically copied into the same file. After several iterations of adding new 'element types' to gather, we ended up with no meaningful (at least to us) data being present in the log file. Perhaps the first lesson we learnt through this process was the need to define a verifiable systematic approach in the future endeavors we were to pursue.

Table 17 presents two fragments (lines 23,416 to 23,427 and 23,587 to 23,593) of the text representation of our source model. In both fragments, information about the *DomainManager* element is defined. Only information in one of those fragments is of interest to us. From the first fragment, lines 23,416 and 23,417 define the *DomainManager* being of type *Class* and the unique identifier of the model element, while lines 23,418 to 23,427 define association relationships with other source model elements. From the second fragment, lines 23,587 to 23,593 refer to physical coordinates of the *DomainManager* element in one specific diagram. While the type of element and

its relationships are of interest to us for structural compliance, the location of such element in a diagram is not.

Table 17. Text representation fragments of the *Spec* model.

Line number	Text representation of source model
...	...
23,416:	(object Class "DomainManager"
23,417:	quid "3C73EAA500FB"
23,418:	used_nodes (list uses_relationship_list
23,419:	(object Uses_Relationship
23,420:	quid "3C6A9814002A"
23,422:	supplier "DomainManagementObjectAddedEventType"
23,423:	quidu "3C6D57460180")
23,424:	(object Uses_Relationship
23,425:	quid "3C6A9819033E"
23,426:	supplier "DomainManagementObjectRemovedEventType"
23,427:	quidu "3C6D574C03E1"))
...	...
23,587:	label (object ItemLabel
23,588:	Parent_View @903
23,589:	Location (950, 468)
23,590:	Nlines 1
23,591:	Max_width 788
23,592:	Justify 0
23,593:	Label "DomainManager")
...	...

Table 18 presents the 'InfoNotConsidered.txt' log file after execution of our tool. It contains the same fragments of the source model that we showed on Table 17, with the difference that the only information left in Table 18 is the one that it is not of interest to us. The information about elements and relationships that we gathered from the source file was used to build a tree structure for later manipulation.

Table 18. InfoNotConsidered.txt log file.

Line number	Text representation of source model
...	...
23,416:	
23,417:	
23,418:	
23,419:	
23,420:	
23,422:	
23,423:	
23,424:	
23,425:	
23,426:	
23,427:	
...	...
23,587:	label (object ItemLabel
23,588:	Parent_View @903
23,589:	Location (950, 468)
23,590:	Nlines 1
23,591:	Max_width 788
23,592:	Justify 0
23,593:	Label "DomainManager")
...	...

After successfully parsing the source model, we started then to implement the first set of mappings that we have defined. Testing the success/failure of our efforts was done by ROSE-RT. The outcome had binary values and was fairly simple to interpret: either the tool was able to successfully read the automatically generated model or not. Once again we relied on automation (ROSE-RT itself) to verify the output of our work. At this point in time we were relying on partially defined test suites and we needed to become more systematic on how to evaluate the contents of the automatically generated intermediate model. The next section addresses such effort.

3.3.2 Transformation Verification Automation

The second force behind our transformation tool was related to the reusability of the mapping definitions. We must emphasize that the reusable aspect of our work points towards reusability of the mapping definitions, and not of our prototype tools that implement them. We needed to provide tools to validate that the mappings were both theoretically and practically sound. It was not sufficient to work only for one

specification, but we looked into the reuse of our work while applying the approach to different standard specifications.

With the reusability aspect in mind, we needed to ensure that every mapping implementation produced the expected results. More specifically, we needed to verify that each mapping was executed when it was defined to do so, as well as not executed when its was not defined to do it. The solution came in the form of what Binder [6, chapter 6] calls a *testable* combinational model represented in the form of truth tables. Table 7 in section 3.1.2 presents an example of such a combinational model. In terms of automation, we moved towards automating support of the use of those tables (activities 2.1 and 2.2 from Figure 55).

We implemented a mechanism to allocate each element of the source model into one of the valid combinations of the testable model. The aim was to be able to verify that at least one element in the source model fit the description of each valid combination and thus verify that we had a complete test suite for each mapping. We soon realized that there were several cases that were not being tested as in the example that follows. Recall from 3.1.1.2 the definition of the ActiveClass mapping.

ActiveClassMapping

Description

Maps an active class from a source model into an active class of the target model.

Context

SWRadio Spec : :Class **source**, ROSE-RT : : Model **target**

Actions

```
//create a corresponding new class in the target model
Class target.c1= new Class( );
target.c1.isActive = TRUE ;
target.c1.name = source.name;
```

Constraints

```
P1:  // The source element is of type Class
     source.oclIsTypeOf(Class) and
P2:  // The source class is an active class
      source.isActive and
P3:  // The class is not a specialization of another element
      (source.generalization.parent->isEmpty( ) or
P4:  // The class is an specialization of one or more elements
      (source.generalization.parent->notEmpty( )
     // For all the generalized elements in which the source element is an specialization
      and (source.generalization.parent -> forAll (p |
P5:  // Generalized elements are of type Class
          (p.oclIsTypeOf(Class) and
P6:  // Generalized elements are active classes
            (p.isActive))))))
```

Recall also from 3.1.2 the ActiveClassMapping testable combinational model that we wanted to provide support for.

Table 19. ActiveClassMapping testable combinational model (valid cases only).

	P1	P2	P3	P4	P5	P6	Result	Comments
1	0	0	0	1	0	0	0	Source element is not a class (not P1), not active (not P2), with parent(s) (not P3 and P4), parent(s) is/are not classes (not P5) and parent(s) is/are not active (not P6).
2	0	0	1	0	0	0	0	Source element is not a class (not P1), not active (not P2), with no parents (P3 and not P4), parent(s) is/are not classes (not P5) and parent(s) is/are not active (not P6).
3	1	0	0	1	0	0	0	Source element is a class (P1), not active (not P2), with parent(s) (not P3 and P4), parent(s) is/are not classes (not P5) and parent(s) is/are not active (not P6).
4	1	0	0	1	1	0	0	Source element is a class (P1), not active (not P2), with parent(s) (not P3 and P4), parent(s) is/are classes (P5) and parent(s) is/are not active (not P6).
5	1	0	0	1	1	1	0	Source element is a class (P1), not active (not P2), with parent(s) (not P3 and P4), parent(s) is/are classes (P5) and parent(s) is/are active (P6).
6	1	0	1	0	0	0	0	Source element is a class (P1), not active (not P2), with no parents (P3 and not P4), parent(s) is/are not classes (not P5) and parent(s) is/are not active (not P6).
7	1	1	0	1	0	0	0	Source element is a class (P1), active (P2), with parent(s) (not P3 and P4), parent(s) is/are not classes (not P5) and parent(s) is/are not active (not P6).
8	1	1	0	1	1	0	0	Source element is a class (P1), active (P2), with parent(s) (not P3 and P4), parent(s) is/are classes (P5) and parent(s) is/are not active (not P6).
9	1	1	0	1	1	1	1	Source element is a class (P1), active (P2), with parent(s) (not P3 and P4), parent(s) is/are classes (P5) and parent(s) is/are active (P6).
10	1	1	1	0	0	0	1	Source element is a class (P1), active (P2), with no parents (P3 and not P4), parent(s) is/are not classes (not P5) and parent(s) is/are not active (not P6).

Considering the mapping definition and the testable combinational model above, we implemented tool support to evaluate if source model elements had the properties (or not) to be considered for mapping purposes. Figure 56 presents a first attempt to define a test model for our example.

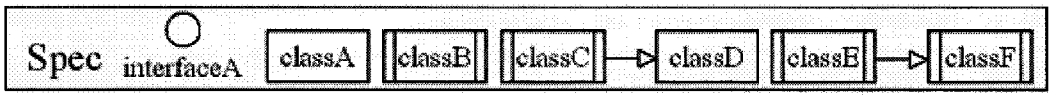


Figure 56. Source model for testing purposes.

Based on the test model from Figure 56, the tool categorized each element onto one of the valid combinations from Table 19. The tool did it by evaluating each of the constraints of a mapping on each of the elements of the test model. Three different values were considered for each property (in the form of a question): Yes, No or Not Applicable (N/A). After evaluation, elements from Figure 56 showed properties as defined in Table 20.

Table 20. Properties of test model elements

	Constraint (property)	InterfaceA	ClassA	ClassB	ClassC	ClassD	ClassE	ClassF
P1	Is the source element a Class?	No	Yes	Yes	Yes	Yes	Yes	Yes
P2	Is the source element an Active Class?	N/A	No	Yes	Yes	No	Yes	Yes
P3	Has the source element 0 parents?	Yes	Yes	Yes	No	Yes	No	Yes
P4	Has the source element 1 or more parents?	No	No	No	Yes	No	Yes	No
P5	Are all the source element parents of type Class?	N/A	N/A	No	Yes	No	Yes	No
P6	Are all the source element parents active classes?	N/A	N/A	N/A	No	N/A	Yes	N/A

The tool performed the evaluation of elements by defining an array of Boolean values for the different properties. Having individual values for each property (constraint) allowed us to use that property value for more than one mapping (i.e., an element having one or more parents might be used in the ActiveClass mapping as in the example, as well as in the Generalization mapping defined in section 4.1.1.3). The tool then assigned a true (1) value for each property with a "Yes" as an answer and assigned a false (0) value for "No" and "N/A" answers. A variation on the representation of Table 20 is shown in Table 21 using values of '0' and '1'.

Table 21. Properties of test model elements (0s and 1s)

Source Element	P1	P2	P3	P4	P5	P6
InterfaceA	0	0	1	0	0	0
ClassA	1	0	1	0	0	0
ClassB	1	1	1	0	0	0
ClassC	1	1	0	1	1	0
ClassD	1	0	1	0	0	0
ClassE	1	1	0	1	1	1
ClassF	1	1	1	0	0	0

Finally, the tool allocated each of the elements from Table 21 into one of the combinations of Table 22. It did it based on the constraints values from Table 21 matching the constraints values from Table 22.

Table 22. ActiveClassMapping verification against the testable combinational model (valid cases only)

	P1	P2	P3	P4	P5	P6	Result	Element
1	0	0	0	1	0	0	0	
2	0	0	1	0	0	0	0	InterfaceA
3	1	0	0	1	0	0	0	
4	1	0	0	1	1	0	0	
5	1	0	0	1	1	1	0	
6	1	0	1	0	0	0	0	ClassA, ClassD
7	1	1	0	1	0	0	0	
8	1	1	0	1	1	0	0	ClassC
9	1	1	0	1	1	1	1	ClassE
10	1	1	1	0	0	0	1	ClassB, ClassF

From Table 22, combinations that did not contain an element were reported as 'not in model' combinations that led us to update our test model. We repeated the process until every combination on Table 22 contained at least one element from the source model.

When applying our transformation tool with a complete set of testable elements, we were able to uncover flaws in the implementation of the mapping definitions. Without this type of verification, the mapping could have proven correct for our current specification, but we could only guess on its application to other standard specifications. We could not have claimed mapping reusability, as we believe we can after enhancing our correctness verification process through automation.

There was still the question of semantic equivalence between the *Spec* and *tSpec* models that the available infrastructure was not able to provide answers for. The next section discusses what we learnt through such a process.

3.3.3 Complete Set of Mapping Definitions

The third and final force behind our transformation automation tools is related to the semantic equivalence of the source and output models. As described earlier in the chapter, we succeeded in automating the creation of an intermediate model. We even verified that our mappings were correctly implemented. But the standard-specific context of our approach required us to go further. We needed to build a tool that not only transformed a model into another model, but also to build the required infrastructure to verify that both models were semantically equivalent. We needed to prove that we were preserving the contents of the original specification model in our aim to express it in a different language. So although apparently the target was only to implement our mapping definitions, we had to enhance our transformation tool to also help us in verifying that every element/relationship in the specification model was used as an originator of an element/relationship in the intermediate model. The outcome allowed us to identify elements and relationships from the spec model that were left behind and not considered for mapping execution. This section describes activity 3.2 from Figure 55.

Recall from Chapter 3 the concept of semantic equivalence. Recall also Figure 23 that presents an abstract representation of the problem at hand. We reproduce Figure 23 from Chapter 3 as Figure 57 in this section. As defined in the example from Chapter 3, models in Figure 57 cannot be considered semantically equivalent because element *E* from the *Spec* is not represented (or was not used as originator of an element) in the *tSpec*.

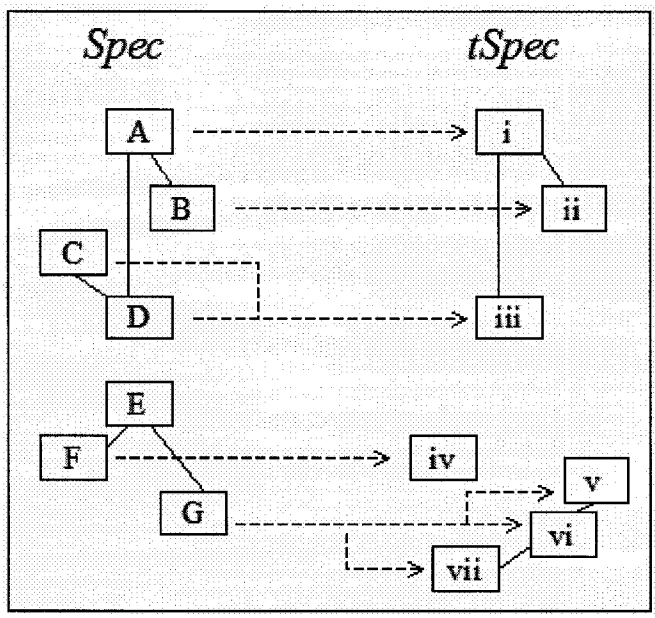


Figure 57. *Spec* elements as originators of *tSpec* elements.

Perhaps a common contribution from our prototype tools to our overall work was the production of information that guided us towards our goals. In the case of identifying the source metamodel, the tool flagged types of elements from the *Spec* model that were not being included in the *Spec* metamodel, and for whom we needed to define mappings to keep the semantic equivalence between *Spec* and *tSpec* models. In the case of verifying the correctness of the mapping implementations, the tool not only gave us a pass or no pass as a result, but also provided us with specific information about the test cases that were being used, and most importantly, about the test cases that were not being tested at all. Finally, when trying to identify which elements/relationships were used (or not) for mapping execution, the tool not only provided us with the identification of those elements/relationships, but also gave us additional information with respect of their types of and location within the *Spec* model. Once the tool was executed, the output data was easily manipulated to identify groups of elements/relationships that were not considered for mapping execution. From our example in Figure 57, our tool counted the number of time an element from the *Spec* model was used for mapping execution. A graphical representation of the result is shown in Figure 58. As we visually saw in the example, most elements from the Spec model were used as originators of elements in the tSpec model. This time, it was the tool that told us that element *E* from the *Spec* model had not

been used for mapping purposes. Only element E was not used (or used 0 times) for mapping execution.

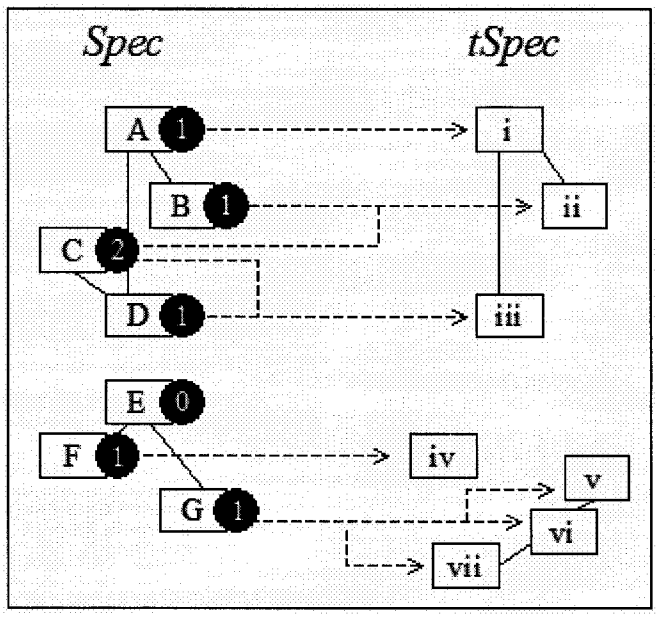


Figure 58. *Spec* elements as originators of *tSpec* elements with tool support.

An additional example of this topic is shown in section 4.1.3 in the next chapter. The next section addresses the forces behind tool support for structural compliance purposes.

3.3.4 Structural Compliance tool Support

As mentioned in section 3.2.1, we use a tool for the *Spec* to *tSpec* transformation and built a separate tool for verification of structural compliance. This section refers to activities 4.1, 4.2, 4.3 and 4.4 from Figure 55

After building the transformation tool, building the compliance verification tool was mostly about reusing what we have already done. In the first tool we defined parsing routines to read the *Spec* model. For the second tool we reused the parsing code and just changed the type of information the tool was going to gather. In the first tool we defined a tree structure to dynamically manipulate the contents of the *Spec* model. For the second tool we reused the same tree structure and the manipulation routines. Activities 4.1 and 4.2 from Figure 55 were accomplished when we were capable of reading the *tSpec* and

Design models. Activity 4.3 from Figure 55 followed the same guidelines as the previous two.

As we explained in Chapter 3, this activity is focused on structural comparison of two models. In other words, elements defined in the first model were looked for in the second. As we also explained before, the complexity of this activity was substantially reduced because we are comparing two models that are written in the same language. The semantic gap between two models written using different metamodels was dealt with in the first three activities of the approach. So structural comparison of two models became a simple activity. Its implementation was also a simple one, but the benefit of having an automated tool to perform such task was of great advantage. We remind the reader that we were dealing with models with hundreds of elements and complex relationships.

Perhaps the most important feature we built in the second tool (that we also reused from the first tool) was the focus on generating information that could guide us on what was left to do to achieve compliance between the *Design* and *tSpec* models. Again, the tool was designed to provide not only information about where the *Design* model was not being compliant with the *tSpec* model, but also about the types and location of the non-compliant *Design* elements/relationships.

We conclude Chapter 3 with a brief summary of what we have achieved in the chapter. Throughout we described at length our proposed two-step approach for compliance verification. The first step to transform the specification model was broken down into three activities. We did so to enable the automatic generation of a semantically equivalent model (we called it *tSpec* model) to be later used for structural and behavioral compliance, which constitute the two activities we propose for the second step in our approach for model compliance. For each activity, we discussed its foundations, proposed a possible solution and, in some cases, showed how automation could contribute to speed up the process and increase the reliability of its outcome. The next chapter presents a case study of the application of the approach to the SWRadio specification.

Chapter 4. Software Radio Case Study

In this chapter we present a case study of the application of our approach to the Software Radio Domain. The case study by itself constitutes one of the main contributions of this dissertation. We pursue the following goals with our case study:

1) Prove the feasibility of the approach. In Chapter 3 we presented the theory behind our approach, with the case study we intend to show its application.
2) Show that the activities we claim can be automated are indeed automatable. As mentioned before, we have built tools as part of the implementation of our approach. In this chapter we show actual results that were produced from those prototype tools.
3) Define mappings that transform a UML 1.5 model into a ROSE-RT model, which includes concepts such as capsules, ports and connectors recently incorporated into UML 2.0 [76]. Mapping definitions are a substantial part of this thesis contribution and although drawn from a software radio specification, we believe that the mappings defined in this chapter may be reused in other domains and for different specifications (see Chapter 5 for a full discussion on the reusability of our mappings)
4) Give standard organizations evidence that specifications models can be used as the main source for verification tools, as opposed to independent programs that try to capture (and later verify) the intent of the specification.

For our case study we use the *Specification for PIM and PSM for Software Radio Components* developed by the OMG Software-Based Communication (SBC) Domain Task Force (DTF). Following the OMG's Model Driven Architecture (MDA) approach, the SBC defines its *Specification for PIM and PSM for Software Radio Components* in the form of a UML PIM and a CORBA PSM. This specification describes high-level

relationships between domain concepts that correspond to the main entities that compose the Software Radio (hereafter SWRadio) architecture and the required interface of those concepts. It uses only conventional UML notation (i.e., mostly class diagrams, with some use cases, sequence diagrams, and very few simple state machines). The Software-Based Communication DTF has submitted for approval its SWRadio PIM so that it can eventually be defined as an OMG Software Defined Radio standard specification.

The case study is presented in two sections. Section 4.1 describes our application of our two-step approach for compliance verification to the first publicly released software radio specification model. Section 4.2 talks about the challenges we face in using a newer version of the Software-Based Communication of the software radio specification.

4.1 Software Radio PIM

The SWRadio PIM we refer throughout this section is the first version of a software radio specification model publicly released from the Software-Based Communication DTF[10]. The SWRadio PIM was introduced in section 2.3, so this section focuses on the application of the 5 activities of the two-step approach for compliance verification using the SWRadio PIM as our case study specification model.

We use independent subsections to present each of the activities of our proposed approach for compliance verification.

4.1.1 SWRadio Semantic validation of Mappings

According to our two-step approach for compliance verification, the first task to do is to create a semantically equivalent representation of the specification model using the same language used to define the design models we want to verify for compliance. To do so, we need to analyze the metamodels used in the definition of both the specification and design models. We then build a bridge in the form of mappings that would transform the specification model into what we call a transformed specification or *tSpec*. We define

[10] The specification has evolved since. We further discuss this topic in section 4.2.

those mappings by establishing semantic relationships between elements of the two metamodels. To do so, we first present the specification and design metamodels. We then present a hierarchy of mappings that allow us to transform the *Spec* model into its semantically equivalent *tSpec*.

4.1.1.1 SWRadio *Spec* metamodel

The first task to do is to analyze the SWRadio PIM as our specification model. The SWRadio PIM was written using UML 1.5. Our aim is to identify the UML modeling elements used in the SWRadio PIM definition rather than avoid unnecessary and time-consuming effort to define mappings for every available element of UML.

Our analysis identifies that the SWRadio PIM is specified using a hierarchy of 43 packages. In these packages, 221 classes and 19 interfaces are defined. Classes and interfaces regroup 112 operation definitions, which together define 84 parameters. Classes and interfaces also define 271 attributes. In terms of relationships, the SWRadio PIM defines 22 simple associations, 20 aggregations, 102 compositions and 60 generalization relationships between classes. There are also 18 realization relationships between classes and interfaces.

The *Spec* metamodel is composed of the elements and relationships used in the SWRadio PIM as outlined in the previous paragraph. We describe each of them. Valid entities and relationships are graphically shown in Table 23.

In the SWRadio PIM, the following type of elements are used:

- Class. Classes define objects that share the same specification of attributes, operations, constraints and semantics. The SWRadio PIM does not differentiate between active and passive classes.
- Interface. Defines a contract that classes realizing the interface must fulfill. (Interfaces are considered declarations and cannot be directly instantiated).

- Package. Defines a placeholder or container used to organize software elements so the system under development is not cluttered with seemingly unrelated elements in the same location.
- Attribute. Defines a structural feature of a classifier that characterizes instances of the classifier. An attribute establishes a named relationship of an instance of a classifier to a value or values.
- Operation. Defines a structural feature that declares a service that a classifier can perform. An operation can have incoming/outgoing parameters and a return value.
- Parameter. Defines an argument that is passed into or/and out of an operation. The type of the parameter restricts what values can be passed.
- Lifeline. Represents individual participants in an interaction. A lifeline can have an instance name, and the role name associated with the instance.
- Message. Defines sending of a signal from one instance to the other. A message may also represent the passing of information from one instance to another within an interaction.

Relationships included in the *Spec* metamodel:

- Aggregation. Defines a weak 'whole-part' relationship between a container and a contained element. In the SWRadio PIM the existence of an aggregation explicitly means that the 'contained' element may be contained in multiple 'containers' and is not tied to the existence of any of its potential containers.
- Composition. Defines a strong 'whole-part' relationship between two elements. This kind of aggregation defines that the lifetime of the 'part' is tightly related to the lifetime of the 'whole'. A 'part' associated with a 'whole' cannot live after the 'whole' is destroyed.
- Dependency. Defines that an element requires other elements for its specification or implementation. In the dependency relationship, one element plays the role of the supplier while the other plays the role of the dependent. The arrowhead points in the direction of the supplier element.

- Generalization. Defines a relationship between a more general classifier and a more specific classifier. The specific classifier in the relationship inherits the properties of the more general classifier.
- Association. Defines a semantic relationship between two or more elements. There is no ownership established between the associated elements. It describes the need for the two elements to communicate or to share some of their properties with each other.

4.1.1.2 ROSE-RT Design metamodel

In this section we present the *Design* metamodel used throughout this case study. The SWRadio Design metamodel is composed of the set of concepts used in the ROSE-RT tool [33] chosen as our design language (graphically shown in Table 23). ROSE-RT is a UML modeling tool for real-time embedded systems. It is used for the development of industrial embedded systems in different application domains including aerospace, telecommunication, and defense. The metamodel defines entities and valid relationships between them. The notation used in ROSE-RT was based on the Real-Time Object Oriented Modeling (ROOM) [92] Modeling Language. The ROOM key concepts: actors, protocol classes, ports and bindings were first introduced in the UML literature in [93] as capsules, protocol classes, ports and connectors respectively. Most recently such concepts have been incorporated into UML 2.0 [76] as structured classes, ports and connectors (although the protocol class concept per se does not exist in UML 2.0, the concept can be expressed by establishing required and provided interfaces that a port must realize).

In ROSE-RT, the following entities are defined:

- Class. Same as in the SWRadio *Spec* metamodel. ROSE-RT additional constraints consider the use of classes only for passive classes. Classes in ROSE-RT are used as data holders.
- Capsule. A capsule is an active class that executes on its own thread of control allowing several capsules to run concurrently. Capsules communicate through

ports. Capsules are elements with the capability to have an internal structure and ports. Capsules are defined as Classes in the Composite Structure package in UML 2.0. The Capsule behavior is defined by means of a hierarchical state machine (statechart).

- Package. Same as in the SWRadio *Spec* metamodel.
- Attribute. Same as in the SWRadio *Spec* metamodel.
- Operation. Same as in the SWRadio *Spec* metamodel.
- Parameter. Same as in the SWRadio *Spec* metamodel.
- Protocol. A Protocol defines the set of messages that can be sent (outgoing messages) or received (incoming messages) through instances of the protocol class (i.e., ports). The protocol concept itself is not defined in UML 2.0. Associated interfaces can constitute a protocol. Associating interfaces is a new concept in UML 2.0 in which interfaces are labeled as 'required' (set of features/operations that an element realizing the interface needs from other entities) or as provided (set of features/operations that an element realizing the interface needs to implement and offer to other entities) [76].
- Signal. Defines an asynchronous stimulus that triggers a reaction in the receiver and without a reply. Signals are capable of carrying a single piece of data attached to them.
- Port. Defines an interaction point between a Capsule and its environment, or between the behavior of a capsule and its contained parts. Ports are connected by connectors. In the SWRadio Design metamodel, ports are defined as instances of protocol classes. Ports are owned by capsules and are the only means of communication from a Capsule with other Capsule or with the environment. The port concept did not exist in previous versions of UML and has been introduced in UML 2.0.
- Lifeline. Represents individual participants in an interaction. A ROSE-RT lifeline can have an instance name, the role name associated with the instance, and the capsule name associated with the role name.

- Message. Defines the passing of information from one instance to another within an interaction. In ROSE-RT, besides holding information about the sender and receiver instances, information about the ports being used to send/receive the message can also be captured. This additional port information is required by Rational Quality Architect (RQA) for compliance testing. A message may also represent the sending of a signal from one instance to the other.

Relationships included in ROSE-RT:

- Aggregation. Same as in the SWRadio *Spec* metamodel. Additional constraints in ROSE-RT consider that a class cannot contain (or aggregate) capsules.
- Composition. Same as in the SWRadio *Spec* metamodel. Additional constraints in ROSE-RT consider that a class cannot contain (or compose) capsules.
- Dependency. Same as in the SWRadio *Spec* metamodel.
- Generalization. Same as in the SWRadio *Spec* metamodel. Additional constraints in ROSE-RT consider that generalization relationships can only be defined between elements of the same type (e.g., a class cannot specialize a capsule nor a capsule specialize a class).
- Association. Same as in the SWRadio *Spec* metamodel.
- Connector. Defines a link between two ports to enable communication between capsules owning the ports.

4.1.1.3 Semantic Validation of Mapping Definitions

In this section we aim to establish semantic relationships between elements of the two metamodels. Table 23 presents a summary of the elements used in the two metamodels and whether they are present in one or the two metamodels.

Table 23. *Spec* and Design model elements.

Graphical icon	Model Element	*Spec*	*Design*
	Class (*Spec*) Data Class (*Design*)	✓	✓
	<<Capsule>>		✓
	Interface	✓	
	Package	✓	✓
	<<Protocol Class>>		✓
	<<Port>>		✓
N/A	Attribute	✓	✓
N/A	Operation	✓	✓
N/A	Parameter	✓	✓
	Lifeline	✓	✓
	Message	✓	✓
	Generalization	✓	✓
	Realization	✓	
	Composition	✓	✓
	Dependency	✓	✓
	Aggregation	✓	✓
	Association [11]	✓	✓
	Connector		✓

Based on the elements above, we establish semantic relationships between elements of the two metamodels. From Table 23 we notice that one element and one relationship from the *Spec* metamodel are not used in the ROSE-RT metamodel. We have defined a mapping to transform interfaces from the *Spec* into ROSE-RT protocol classes, as well as a mapping to transform interface realizations into protocol class compositions for port creation. Later in this chapter we discuss in detail how we define such transformations, as well as the rest of our *Spec* to ROSE-RT mapping definitions.

[11] Although associations and connectors use the same graphic icon, in a diagram they can be identified by the type of element they link: associations links two classifiers while connectors link two ports.

Transformation of a *Spec* model into a *Design* model is not a trivial issue. Many different aspects contribute to create a gap between the source and target models. In our research work, the two models start close to the other when we only consider semantically related entities from both models. The gap grows when we consider how those entities relate to each other in the *Spec* model, and how each of those relationships is to be expressed into the *Design* model. Different interobject communication paradigm (procedure call based on *Spec* vs. signal based on *Design*) widens the semantic gap further apart. To address the change in the interobject communication approach, there is the need to create elements in the *tSpec* model that do not exist in the *Spec* model. Finally, the fact that additional information is required to transform a *Spec* into a *tSpec* suggests that we are dealing with a non-trivial issue. We define individual mappings that consider each of the previous aspects to bridge the gap between *Spec* and *Design* models.

In defining our mappings, we consider the necessity of each individual mapping, as well as sufficiency of the whole set of mappings. Necessity of each mapping is addressed by the fact that we want every element/relationship from the *Spec* model to be used as an originator of an element/relationship of the *tSpec* model. For each type of element/relationship in the *Spec* model, there is the need to define a mapping so the *Spec* element/relationship is represented in the *tSpec* model. The sufficiency aspect of our mappings is addressed by activity 3 of our approach: completeness of mapping definitions. We explained activity 3 in section 3.1.3 and provide a SWRadio example later in 4.1.3.

Based on the aspects described in the previous paragraph, we have grouped our mapping definitions into four different categories:

1) **Semantically Related Element Mappings**. Mappings that take elements of the *Spec* model and copy them into semantically equivalent elements in the *Design* model.
2) **Semantically Related Relationship Mappings**. Mappings that take relationship elements from the *Spec* model and copy them into semantically equivalent relationship elements in the *Design* model.

3) **Interobject Communication Paradigm Mappings.** As described above, the interobject communication paradigms used in the source and target models is different. The *Spec* model uses a procedure call based type of communication while the *Design* model uses message (signal) based type of communication. The transformation of one paradigm to the other involves the creation of protocols, signals and ports in the *tSpec* model, elements that do not exist in the *Spec* model. This type of mappings bridges the semantic gap between the two metamodels with respect to its interobject communication paradigm.
4) **Additional Transformation Input Mappings.** The Transformation Input model provides additional information that cannot be found in the *Spec* model. The application of mappings included in this group depends on the information defined in the Transformation Input model. Most of the mapping constraints refer to this model to evaluate if the mapping is to be applied or not (this type of mappings might also be found in a different group of mappings).

Tables 24 through 26 present mapping lists for each of the first three groups. The first column contains the name of the mapping. The second column shows the element/relationship from the *Spec* model to be mapped. The third and final column presents the element/relationship in the *tSpec* model as a result of the mapping. The basic reading of the table would be: "from the *Spec* model map elements/relationships defined in the second column into the *tSpec* model as elements/relationships of the third column"

Table 24. Semantically related element mappings

Semantically Related Element Mapping	*Spec* Element	*tSpec* Element
AttributeMapping	Attribute	Attribute
ClassToCapsuleMapping	Class	Capsule
ClassToDataClassMapping	Class	Data class
InteractionInstanceMapping	Interaction Instance	Interaction Instance
InteractionMapping	Interaction	Interaction
InterfaceToProtocolClassMapping	Interface	Protocol
MessageMapping	Message	Message
OperationMapping	Operation	Operation
PackageMapping	Package	Package
ParameterMapping	Parameter	Parameter

Table 25. Semantically related relationship mappings

Semantically Related Relationship Mapping	*Spec* Relationship	*tSpec* Relationship
AggregationMapping	Aggregation	Aggregation
AssociationMapping	Association	Association
CompositionMapping	Composition	Composition
DependencyMapping	Dependency	Dependency
GeneralizationMapping	Generalization	Generalization

Table 26. Interobject communication paradign mappings

Interobject Communication Paradigm Element mapping	*Spec* Element	*tSpec* Element
AssociationToPortMapping	Association	Port
AssociationToProtocolMapping	Association	Protocol
DataClassToSignalTypeMapping	Class	Signal Type
DataClassToArgumentTypeMapping	Class	Argument Type
OperationResultToSignalMapping	Operation	Signal
OperationToDataClassMapping	Operation	Data Class
OperationToSignalMapping	Operation	Signal
ParameterToAttributeMapping	Parameter	Attribute
ParameterTypeToSignalTypeMapping	Parameter type	Signal Type

Table 27 presents a list of mappings that are directly influenced by the additional input provided from the designer. These mappings are only applied if the name of the element/relationship instance being mapped is found in the list provided in the first column. The second column presents the category and the name of the mapping. The third column shows the element/relationship from the *Spec* model to be mapped. The fourth and final column presents the element/relationship in the *tSpec* model as a result of the mapping. The basic reading of the table would be: "from the *Spec* model, map elements/relationships defined in the third column into the *tSpec* model as elements/relationships of the fourth column.

Table 27. Transformation input constrained mappings

Transformation Input List	Category/Mapping	*Spec* Element	*tSpec* Element
	Communication paradigm		
Association list	AssociationToConnectorMapping	Association	Connector
Capsule list	ClassToCapsuleMapping	Class	Capsule
Operation list	OperationToSignalMapping	Operation	Signal
	OperationResultTypeToSignalTypeMapping	Operation	Signal type
	OperationToDataClassMapping	Operation	Data class
	ParameterToAttributeMapping	Parameter	Attribute
	DataClassToSignalTypeMapping	DataClass	Signal type
	Semantically related element mapping		
Type correspondence	AttributeMapping	Attribute	Attribute
	OperationResultToSignalMapping	Operation	Signal
	ParameterToAttributeMapping	Parameter	Attribute
	ParameterToSignalMapping	Parameter	Signal

Figure 59 presents the SpecToRoseRT mapping hierarchy. It shows a graphical grouping per category of mappings, and also shows a mapping dependency that has been established between them. For example, ParameterToAttribute (bottom left of the figure) mapping can only be invoked while executing OperationToDataClass mapping, which in turn, may be invoked while executing OperationToSignal mapping.

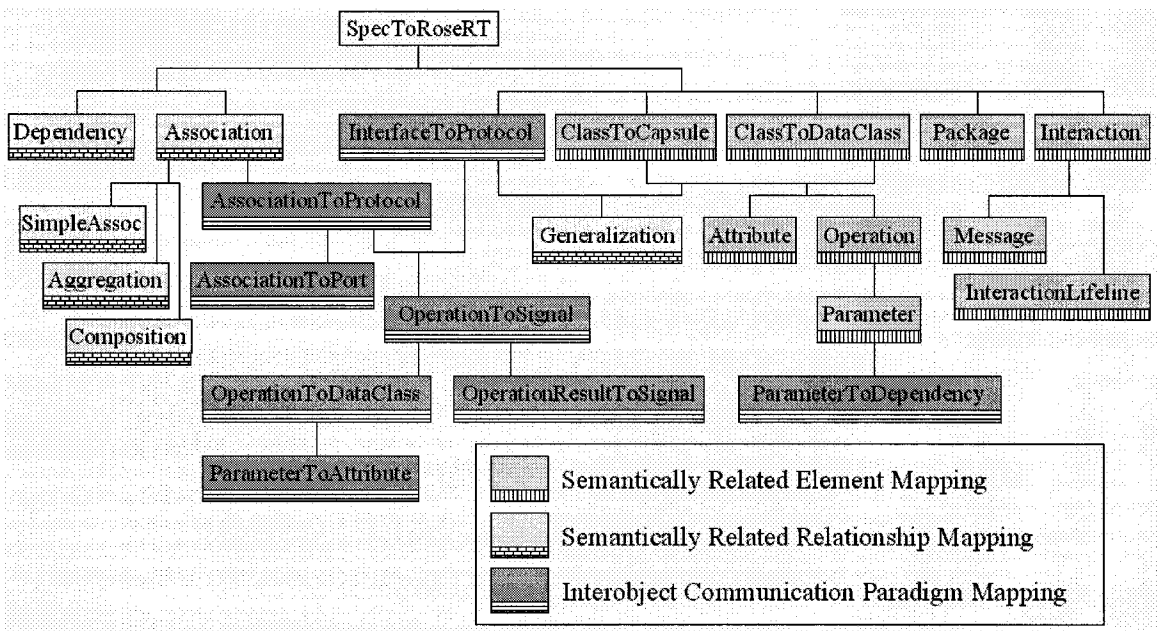


Figure 59. SpecToRoseRT mapping hierarchy

Table 28 presents the number of times every mapping was applied while transforming the SWRadio PIM (*Spec*) into our *tSpec* model.

Table 28. Number of mapping applications in SWRadio PIM (*Spec*) to *tSpec* transformation

	Mapping	#
1	AssociationToProtocolMapping	1
2	AggregationMapping	21
3	AssociationMapping	103
4	AssociationToPortMapping	61
5	AssociationToProtocolMapping	17
6	AttributeMapping	271
7	ClassToCapsuleMapping	14
8	ClassToDataClassMapping	210
9	CompositionMapping	98
10	DependencyMapping	53
11	GeneralizationMapping	54
12	InteractionLifeLineMapping	23
13	InteractionMapping	4
14	InteractionMessageMapping	33
15	InterfaceToProtocolMapping	19
16	OperationMapping	55
17	OperationResultToSignalMapping	20
18	OperationToDataClassMapping	10
19	OperationToSignalMapping	59
20	PackageMapping	42
21	ParameterMapping	38
22	ParameterToAttributeMapping	22
23	ParameterToDependencyMapping	90
	Total	1318

We conclude this section with the detailed definition of mappings using the format presented in 3.1.1. The reader who is not interested in such exhaustive descriptions is invited to skip to the next section. We also point the reader to section 4.1.2 where we discuss how testing was performed to verify that the mappings (to be presented below) produced the intended output.

Considering the possibility of future reuse of our mapping definitions we have included (in the description section of each mapping) a brief discussion that associates the mapping with one of the following groups:

1) *Case Study.* Mappings under this group require information that is particular to our case study and its applicability to other model transformations most probably would imply the modification of the mapping to adapt it to a new context.

2) *UML 1.x to ROSE-RT.* This group of mappings might be reused when transforming models from a UML 1.x model to the ROSE-RT language. This type of mappings does not require case study specific information.
3) *General UML.* This group of mappings is free of case study specific information, and also independent from the ROSE-RT language.

AggregationMapping (from Semantically Related Relationship Mapping)

Description

Maps a SWRadio Spec::*Association* from the source model into a *ROSE-RT::Association* in the target model. Aggregations in both metamodels are defined through the *aggregationKind* attribute of the *memberEnds* of *Associations.* We consider this mapping to be in the *Case Study* group of mappings. The constraints (P3 and P4) defined for this mapping require information about the activeness/passiveness of the associated classes from the source model. The *Spec* model that constitutes our source model does not include such information, and we have proposed to gather it from an additional model that we have called *Transformation Input* model. The latter causes the mapping to be highly coupled to our case study. Having a source model that includes information about the activeness/passiveness of a class would make us to consider the mapping within the *General UML* mapping group. There is also the possibility of removing the constraints with respect of the activeness/passiveness of the associated classes, and that would also make the mapping to be considered in the *General UML* group of mappings. Considering both of the previous cases, the mapping can be used whenever an aggregation relationship from a source model wants to be copied as an aggregation relationship in a target model.

Context

SWRadio Spec::Association **a**, Transformation Input : :ActiveClass **ac[ ]**, ROSE-RT : : Model **t**

Actions

```
\\ create a new association
Association a1 = new Association ( );
\\ assign memberEnd properties to the new association
a1.memberEnd->at(1)= a.memberEnd->at(1);
a1.memberEnd->at(2)= a.memberEnd->at(2);
\\ Add association to the target model
t.add(lookupPathName (a), a1);
```

Constraints

Invoked from **AssociationMapping**

P1 \\ Source element is an association
a.oclIsTypeOf(Association) **and**
P2 \\ Only one member is aggregated (the 'whole')
a.memberEnd->select (aggregationKind = 'shared') -> size ( ) = 1 **and**
P3 \\ 'Whole' classifier is not an active class
if (ac->select(name=a.memberEnd->select(aggregationKind='shared').name)->isEmpty( ))
P4 \\ 'Part' classifier is not an active class
ac->select(name=a.memberEnd->select(aggregationKind='none').name)->isEmpty( )
Endif

AssociationMapping (from Semantically Related Relationship Mapping)

Description

Maps a *SWRadio Spec::Association* from the source model into a *ROSE-RT::Association* in the target model. We consider this mapping to be in the *Case Study* group of mappings. This mapping requires information about the activeness/passiveness of the associated classes from the source model. The *Spec* model that constitutes our source model does not include such information, and we have proposed to gather it from an additional model that we have called *Transformation Input* model. The latter causes the mapping to be highly coupled to our case study. Having a source model that includes information about the activeness/passiveness of a class would make us to consider the mapping within the *General UML* mapping group. In such a case, it would also be required to remove the call to the AssociationToProtocolMapping in the end of the action section because in our definition protocols are specific to the ROSE-RT metamodel.

Context

SWRadio Spec : :Association **a**, Transformation Input : :ActiveClass **ac[ ]**, ROSE-RT : : Model **t**

Actions

```
\\ End 1 is not an active class
if ( (ac->select(name=a.memberEnd->at(1).name) ->isEmpty() and
   ac->select(name=a.memberEnd->at(2).name) ->isEmpty() and
   a.memberEnd->at(1).aggregationKind='none' and
   a.memberEnd->at(2).aggregationKind='none')
      \\ create a new association
      Association a1 = new Association ( );
      \\ assign memberEnd properties to the new association
      a1.memberEnd->at(1)= a.memberEnd->at(1);
      a1.memberEnd->at(2)= a.memberEnd->at(2);
      \\ Add association to the target model
      t.add(lookupPathName (a), a1);
endif
\\ invoke other mapping(s)
call AggregationMapping;
call CompositionMapping;
call AssociationToProtocolMapping;
```

Constraints

Invoked from **SpecToRoseRTModelMapping**

P1 \\ Source element is an association
a.oclIsTypeOf(Association)

AssociationToPortMapping (from Interobject Communication Mapping)

Description

Maps a *SWRadio Spec::Association* from the source model into a *ROSE-RT::Port* in the target model. We consider this mapping to be within the *UML 1.x to ROSE-RT* group of mappings. In our definition, ports are specific to the ROSE-RT metamodel. The mapping also requires additional elements being present in the target model (i.e. capsules and protocols) that makes the mapping to be highly coupled to the ROSE-RT metamodel and thus makes it reusable only when considering a ROSE-RT model as target.

Context

SWRadio Spec::Association **a**, Transformation Input: :ActiveClass **ac[ ]**, ROSE-RT::Model **t**

Actions

```
\\ Create new ports using the associated class names and role name as part of the port name.
Port p1 = new Port("To_"+ a.memberEnd->at(2).type.name+"_As_"+ a.memberEnd->at(2).name);
Port p2 = new Port("To_"+ a.memberEnd->at(1).type.name+"_As_"+ a.memberEnd->at(1).name);
\\ Assign protocol names
p1.protocol.name = a.memberEnd->at(1).type.name"_TO_"+ a.memberEnd->at(2).type.name;
p2.protocol.name = a.memberEnd->at(1).type.name"_TO_"+ a.memberEnd->at(2).type.name;
\\ Assign the protocol roles to memberEnds of the association
p1.protocolRole ='base';
p2.protocolRole ='conjugate';
\\ select capsules from memberEnds
Capsule c1= t->select(name= lookUpPathName(a.memberEnd->at(1).type);
Capsule c2= t->select(name= lookUpPathName(a.memberEnd->at(2).type);
\\ Add port to capsules
c1.add(p1);
c2.add(p2);
```

Constraints

Invoked from **AssociationToProtocolMapping**

P1 \\ Source element is an association
a.oclIsTypeOf(Association)

AssociationToProtocolMapping (from Interobject Communication Mapping)

Description

Maps a *SWRadio Spec::Association* from the source model into a *ROSE-RT::Protocol* in the target model. We consider this mapping to be in the *Case Study* group of mappings. The constraints (P2 and P3) defined for this mapping require information about the activeness/passiveness of the associated classes from the source model. The *Spec* model that constitutes our source model does not include such information, and we have proposed to gather it from an additional model that we have called *Transformation Input* model. The latter causes the mapping to be highly coupled to our case study. Having a source model that includes information about the activeness/passiveness of a class would make us to consider the mapping within the *UML 1.x to ROSE-RT* mapping group. In such a case the mapping would not be considered within the General UML group because in our definition protocols are specific to the ROSE-RT metamodel.

Context

SWRadio Spec : :Association **a**, Transformation Input : :ActiveClass **ac[ ]**, ROSE-RT : : Model **t**

Actions

```
\\ Create a new protocol using the names of the associated elements
Protocol p1 = new Protocol (a.memberEnd->at(1).type.name"_TO_"+ p.memberEnd->at(2).type.name);
\\ Add protocol to the target model
t.add(lookupPathName (a), p1);
\\ invoke other mapping(s)
call AssociationToPortMapping;
call OperationToSignalMapping;
```

Constraints

Invoked from **AssociationMapping**

P1 \\ Source element is an association
a.oclIsTypeOf(Association) **and**

P2 \\ End 1 is an active class
ac->select(name=a.memberEnd->at(1).name) ->notEmpty() **and**

P3 \\ End 2 is an active class
ac->select(name= a.memberEnd->at(2).name) ->notEmpty()

AttributeMapping (from Semantically Related Element Mapping)

Description

Maps a *SWRadio Spec::Attribute* from the source model into a *ROSE-RT::Attribute* in the target model. We consider this mapping to be in the *General UML* mapping group. Attributes are elements defined in the UML specification and are not particular to our case study or the ROSE-RT metamodel.

Context

SWRadio Spec : :Attribute **a**, ROSE-RT : : Model **t**

Actions

```
\\ Create a new attribute with the name of source element as parameter
Attribute a1= new Attribute (a.name);
\\ Assign the type of the source attribute to the newly created attribute
a1.type = a.type;
\\ Add attribute to the target model in the corresponding classifier
t.add(lookupPathName(a), a1);
```

Constraints

Invoked from **ClassToDataClassMapping or ClassToCapsuleMapping**

P1 \\ Source element is a property (attribute)
a.oclIsTypeOf(Property) **and**

P2 \\ The attribute does not belong to an interface
not a.parent.oclIsTypeOf(Interface)

ClassToCapsuleMapping (from Semantically Related Element Mapping)

Description

Maps a *SWRadio Spec::Class* from the source model into a *ROSE-RT::Capsule* in the target model. The constraint (P2) defined for this mapping requires information about the activeness/passiveness of the source model's class. The *Spec* model that constitutes our source model does not include such information, and we have proposed to gather it from an additional model that we have called *Transformation Input* model. The latter causes the mapping to be highly coupled to our case study. Having a source model that includes information about the activeness/passiveness of a class would make us to consider the mapping within the *UML 1.x to ROSE-RT* mapping group. In such a case the mapping would not be considered within the General UML group because in our definition capsules are specific to the ROSE-RT metamodel.

Context

SWRadio Spec ::Class **c**, Transformation Input ::ActiveClass **ac[ ]**, ROSE-RT :: Model **t**

Actions

```
\\ Create a new capsule with the name of source element as parameter
Capsule c1= new Capsule (c.name);
\\ Add capsule to the target model in the corresponding package
t.add(lookupPathName(c), c1);
\\ invoke other mapping(s)
call AttributeMapping;
call OperationMapping;
call GeneralizationMapping;
```

Constraints

Invoked from **SpecToRoseRTModelMapping**

P1 \\ Source element is a class
c.oclIsTypeOf(Class) **and**

P2 \\ Class designated as active in transformation input model
ac->select (name= c.name)->notEmpty ()

ClassToDataClassMapping (from Semantically Related Element Mapping)

Description

Maps a *SWRadio Spec::Class* from the source model into a *ROSE-RT::Class* in the target model. The constraint (P2) defined for this mapping requires information about the activeness/passiveness of the source model's class. The *Spec* model that constitutes our source model does not include such information, and we have proposed to gather it from an additional model that we have called *Transformation Input* model. The latter causes the mapping to be highly coupled to our case study. Having a source model that includes information about the activeness/passiveness of a class would make us to consider the mapping within the General UML group.

Context

SWRadio Spec : :Class **c**, Transformation Input : :ActiveClass **ac[ ]**, ROSE-RT : : Model **t**

Actions

```
\\ Create a new class in the target model with name of source element as parameter
Class c1= new Class (c.name);
\\ Add class to the target model in the corresponding package
t.add(lookupPathName(c), c1);
\\ invoke other mapping(s)
call AttributeMapping;
call OperationMapping;
call GeneralizationMapping;
```

Constraints

Invoked from **SpecToRoseRTModelMapping**

P1 \\ Source element is a class
c.oclIsTypeOf(Class) **and**

P2 \\ Class is not designated as active in transformation input model
ac->select (name= c.name)->isEmpty ( )

CompositionMapping (from Semantically Related Relationship Mapping)

Description

Maps a *SWRadio Spec::Association* from the source model into a *ROSE-RT::Association* in the target model. Compositions in both metamodels are defined through the *aggregationKind* attribute of the *memberEnds* of *Associations*. We consider this mapping to be in the *Case Study* group of mappings. The constraints (P3 and P4) defined for this mapping require information about the activeness/passiveness of the associated classes from the source model. The *Spec* model that constitutes our source model does not include such information, and we have proposed to gather it from an additional model that we have called *Transformation Input* model. The latter causes the mapping to be highly coupled to our case study. Having a source model that includes information about the activeness/passiveness of a class would make us to consider the mapping within the *General UML* mapping group. There is also the possibility of removing the constraints with respect of the activeness/passiveness of the associated classes, and that would also make the mapping to be considered in the *General UML* group of mappings. Considering both of the previous cases, the mapping can be used whenever a composition relationship from a source model wants to be copied as a composition relationship in a target model.

Context

SWRadio Spec : :Association **a**, Transformation Input : :ActiveClass **ac[ ]**, ROSE-RT : : Model **t**

Actions

```
\\ create a new association
Association a1 = new Association ( );
\\ assign memberEnd properties to the new association
a1.memberEnd->at(1)= a.memberEnd->at(1);
a1.memberEnd->at(2)= a.memberEnd->at(2);
\\ Add association to the target model
t.add(lookupPathName (a), a1);
```

Constraints

Invoked from **AssociationMapping**

P1 \\ Source element is an association
a.oclIsTypeOf(Association) **and**

P2 \\Only one member is composite(the 'whole')
a.memberEnd->select (aggregationKind = 'composite') -> size ( ) = 1 **and**

P3 \\ 'Whole' classifier is not an active class
if (ac->select(name=a.memberEnd->select(aggregationKind='composite').name)->isEmpty( ))

P4 \\ 'Part' classifier is not an active class
ac->select(name=a.memberEnd->select(aggregationKind='none').name)->isEmpty( )
Endif

DependencyMapping (from Semantically Related Relationship Mapping)

Description

Maps a *SWRadio Spec::Dependency* from the source model into a *ROSE-RT::Dependency* in the target model. We consider this mapping to be in the *General UML* mapping group. Dependency relationships are elements defined in the UML specification and are not particular to our case study or the ROSE-RT metamodel.

Context

SWRadio Spec : :Dependency **d**, ROSE-RT : : Model **t**

Actions

```
\\ Create a new dependency relationship
Dependency d1= new Dependency ();
\\ Assign supplier element to supplier role of the Dependency
d1.supplier = d.supplier;
\\ Assign the client element to the client role of the Dependency
d1.client = d.client;
\\ Add the new dependency relationship to the target model
t.add(lookupPathName (d), d1);
```

Constraints

Invoked from **SpecToRoseRTModelMapping**

P1 \\ Source element is a dependency relationship
p.oclIsTypeOf(Dependency)

GeneralizationMapping (from Semantically Related Relationship Mapping)

Description

Maps a *SWRadio Spec::Generalization* from the source model into a *ROSE-RT::Generalization* in the target model. We consider this mapping to be in the *Case Study* group of mappings. The constraint (P5) defined for this mapping requires information about the activeness/passiveness of the associated classes from the source model. The *Spec* model that constitutes our source model does not include such information, and we have proposed to gather it from an additional model that we have called *Transformation Input* model. The latter causes the mapping to be highly coupled to our case study. Having a source model that includes information about the activeness/passiveness of a class would make us to consider the mapping within the *General UML* mapping group. There is also the possibility of removing the constraint with respect of the activeness/passiveness of the generalized classes, and that would also make the mapping to be considered in the *General UML* group of mappings. Considering both of the previous cases, the mapping can be used whenever a generalization relationship from a source model wants to be copied as a generalization relationship in a target model. We highlight that the mapping is defined to only map generalization relationships between elements of the same type. If the more general case does not require such a constraint, then the constraint would need to be removed.

Context

SWRadio Spec::Generalization **g**, Transformation Input::ActiveClass **ac[ ]**, ROSE-RT::Model **t**

Actions

```
\\ create a new generalization
Generalization g1 = new Generalization ( );
\\ Assign the generalized element to the generalization
g1.general= g.general;
\\ Find the specialized element in the target model
Classifier c1=t.lookupPathName(g.specific);
\\ Add generalization to the specialized element
c1.add(g1);
```

Constraints

Invoked from **ClassToDataClassMapping or ClassToCapsuleMapping or InterfaceToProtocolMapping**

P1 \\ Source element is a generalization relationship
g.oclIsTypeOf(Generalization) **and**
P2 \\ target metamodel allows only generalizations between similar type of classifiers
g.specific.type = g.general.type **and**
P3 \\ classifier with more than one parent
if (g.specific.generalization.parent->size ( ) > 1)
P4 \\ classifier is a class
g.specific.oclIsTypeOf(Class) **and**
P5 \\ Classifier not an active class. Target metamodel does not allow multiple inheritance on active classes
ac->select (name= g.specific.name)->isEmpty ( )
Endif

InteractionMapping (from Semantically Related Element Mapping)

Description

Maps a *SWRadio Spec::Interaction* from the source model into a *ROSE-RT::Interaction* in the target model. We consider this mapping to be in the *General UML* mapping group. Interactions are defined in the UML specification and are not particular to our case study or the ROSE-RT metamodel.

Context

SWRadio Spec::Interaction **i**, ROSE-RT::Model **t**

Actions

\\ Create a new interaction with the name of source element as parameter
Interaction i1= new Interaction (i.name);
\\ Add interaction to the target model in the corresponding package
t.add(lookupPathName(i), i1);
\\ invoke other mapping(s)
call InteractionLifeLineMapping;
call MessageMapping;

Constraints

Invoked from **SpecToRoseRTModelMapping**

P1 \\ Source element is an interaction
p.oclIsTypeOf(Interaction)

InteractionLifelineMapping (from Semantically Related Element Mapping)

Description

Maps a *SWRadio Spec::Lifeline* from the source model into a *ROSE-RT::Lifeline* in the target model. We consider this mapping to be in the *General UML* mapping group. Interaction lifelines are defined in the UML specification and are not particular to our case study or the ROSE-RT metamodel.

Context

SWRadio Spec::Lifeline **l**, ROSE-RT::Model **t**

Actions

\\ Create a new lifeline with the name of source element as parameter
Lifeline l1= new Lifeline (l.name);
\\ Find the interaction that will own the new lifeline
Interaction i = t.lookupPathName(l.interaction);
\\ Add lifeline to interaction
i.add(l1);

Constraints

Invoked from **InteractionMapping**

P1 \\ Source element is a lifeline
l.oclIsTypeOf(Lifeline) **and**

InteractionMessageMapping (from Semantically Related Element Mapping)

Description

Maps a *SWRadio Spec::Message* from the source model into a *ROSE-RT::Message* in the target model. We consider this mapping to be in the *General UML* mapping group. Interaction messages are defined in the UML specification and are not particular to our case study or the ROSE-RT metamodel.

Context

SWRadio Spec::Message **m**, ROSE-RT::Model **t**

Actions

```
\\ Create a new message with the name of source element as parameter
Message m1= new Message (m.name);
\\ Assign the sender instance of the message
m1.sendEvent = m.sendEvent;
\\ Assign the receiver instance of the message
m1.receiveEvent = m.receiveEvent;
\\ Find the association relationship between sender and receiver
Relationship assoc = t.lookup(m.sendEvent, m.receiveEvent)
\\ Find and assign sender and receiver's ports
m1.sendPort= t.lookupPathName("To_"+assoc.target+"_As_"+assoc.targetRole);
m1.receivePort= t.lookupPathName("From_"+assoc.source+"_As_"+assoc.sourceRole);
\\ Find the interaction that will own the new message
Interaction I = t.lookupPathName(m.interaction);
\\ Add message to the interaction
I.add(m1)
```

Constraints

Invoked from **InteractionMapping**

P1 \\ Source element is a message
m.oclIsTypeOf(Message)

InterfaceToProtocolMapping (from Semantically Related Element Mapping)

Description

Maps a *SWRadio Spec::Interface* from the source model into a *ROSE-RT::Protocol* in the target model. We consider this mapping to be within the *UML 1.x to ROSE-RT* group of mappings. In our definition, protocols are specific to the ROSE-RT metamodel and thus make it reusable only when considering a ROSE-RT model as target.

Context

SWRadio Spec::Interface **i**, ROSE-RT::Model **t**

Actions

```
\\ Create a new protocol with the name of source element as parameter
Protocol p1= new Protocol (i.name);
\\ Add protocol to the target model in the corresponding package
t.add(lookupPathName(i), p1);
\\ invoke other mapping(s)
call InterfaceToProtocolMapping;
call GeneralizationMapping;
```

Constraints

Invoked from **SpecToRoseRTModelMapping**

P1 \\ Source element is an interface
p.oclIsTypeOf(Interface)

OperationMapping (from Semantically Related Element Mapping)

Description

Maps a *SWRadio Spec::Operation* from the source model into a *ROSE-RT::Operation* in the target model. We consider this mapping to be in the *General UML* mapping group. Operations are defined in the UML specification and are not particular to our case study or the ROSE-RT metamodel.

Context

SWRadio Spec::Operation **o**, ROSE-RT::Model **t**

Actions

```
\\ Create a new operation element with name of source element as parameter
Operation o1= new Operation (o.name);
if (o1.type->notEmpty ( ) )
   \\ Assign the result type if any
   o1.type.name = o1.type.name;
endif
\\ Add operation to the target model
t.add(lookupPathName (o), o1);
\\ invoke other mapping(s)
call parameterMapping ;
```

Constraints

Invoked from **ClassToDataClassMapping or ClassToCapsuleMapping**

P1 `\\ Source element is an operation`
`o.oclIsTypeOf(Operation)` **and**

P2 `\\ The operation does not belong to an interface`
`not o.parent.oclIsTypeOf(Interface)`

OperationResultToSignalMapping (from Interobject Communication Mapping)

Description

Maps a *SWRadio Spec::Operation::Type* from the source model into a *ROSE-RT::Signal* in the target model. The mapping requires information about which operations of the source model are to be mapped into signals on the target model. The *Spec* model that constitutes our source model does not include such information, and we have proposed to gather it from an additional model that we have called *Transformation Input* model. The latter causes the mapping to be highly coupled to our case study. Having a source model that includes information about which operations to map into signals would make us to consider the mapping within the *UML 1.x to ROSE-RT* group of mappings. In our definition, signals are specific to the ROSE-RT metamodel and thus make it reusable only when considering a ROSE-RT model as target.

Context

SWRadio Spec::Operation **o**, Transformation Input::CollaborationOperation **op[ ]**, ROSE-RT::Model **t**

Actions

```
\\ Create a new signal with the name of source element as parameter
Signal s1= new Signal (o.name+"_RESULT");
\\ Find operation involved in collaboration in the Transformation Input model
OperationList op1 = op ->select(name=o.name);
\\ Assign direction of the signal, based on the role of the associated classes:
if (o.class.name== op1.ownerClass)
   \\ incoming messages are defined for target classes
   s1.direction="OUT";
else
    \\ outgoing messages are defined for source classes
    s1.direction="IN";
endif;
\\ Find the protocol in the target model
Protocol p1= t->select(name= op1. sourceClass.name+"_To_"+ op1. TargetClass.name);
\\ Add signal to the protocol
p1.add(s1);
```

Constraints

Invoked from **OperationToSignalMapping**

P1 \\ Source element is an operation
o.oclIsTypeOf(Operation) **and**

P2 \\ Operation has a return value
o.type->notEmpty ( )

OperationToDataClassMapping (from Interobject Communication Mapping)

Description

Maps a *SWRadio Spec::Operation* from the source model into a *ROSE-RT::DataClass* in the target model. We consider this mapping to be within the *UML 1.x to ROSE-RT* group of mappings. In our definition, the creation of data classes from an operation definition is done with the purpose of attaching an instance of the data class to a signal. Also in our definition, signals are specific to the ROSE-RT metamodel and thus make it reusable only when considering a ROSE-RT model as target.

Context

SWRadio Spec::Operation **o**, ROSE-RT::Signal **s** , ROSE-RT::Model **t**

Actions

```
\\ Create a new class using the name of source element
Class c1 = new Class (o.name+"DATA_CLASS_FROM"+o.class.name);
\\ Add class to target model
t.add(lookupPathName(o.parent), c1);
\\ Set the type of the signal to the new data class
s.type=c1.name
\\ invoke other mapping(s)
call ParameterToAttributeMapping;
```

Constraints

Invoked from **OperationToSignalMapping**

P1 \\ Source element is an operation
o.oclIsTypeOf(Operation) **and**

P2 \\ Create class only when operation has 2 or more parameters
o.parameter->size( ) > 1

OperationToSignalMapping (from Interobject Communication Paradigm Mapping)

Description

Maps a *SWRadio Spec::Operation* from the source model into a *ROSE-RT::Signal* in the target model. The mapping requires information about activeness/passiveness of classes as well as which operations of the source model are to be mapped into signals on the target model. The *Spec* model that constitutes our source model does not include such information, and we have proposed to gather it from an additional model that we have called *Transformation Input* model. The latter causes the mapping to be highly coupled to our case study. Having a source model that includes information about activeness/passiveness of classes and also about which operations to map into signals would make us to consider the mapping within the *UML 1.x to ROSE-RT* group of mappings. In our definition, signals are specific to the ROSE-RT metamodel and thus make it reusable only when considering a ROSE-RT model as target.

Context

SWRadio Spec::Operation **o**, Transformation Input::CollaborationOperation **op[ ]**,
Transformation Input::ActiveClass **ac[ ]**, ROSE-RT::Model **t**

Actions

```
\\ Create a new signal with the name of source element as parameter
Signal s1= new Signal (o.name);
\\ If the owner of Operation Is an Interface
if (op.interface->notEmpty () )
   s1.direction="IN";
\\ If the owner of the operation is a class involved in a collaboration
else
   \\ Find operation involved in collaboration in Transformation Input  model
   OperationList op1 = op ->select(name=o.name);
   \\ Assign direction of the signal, based on the role of the associated classes
   if (o.class.name== op1.ownerClass)
      \\ incoming messages are defined for source classes
      s1.direction="IN";
   else
      \\ outgoing messages are defined for target classes
      s1.direction="OUT";
   endif;
endif;
\\ Find the protocol in the target model
Protocol p1= t->select(name= op1.SourceClass.name+"_To_"+ op1. targetClass.name);
\\ Add signal to the protocol
p1.add(s1);
\\ invoke other mapping(s)
call OperationResultToSignalMapping;
call OperationToDataClassMapping;
```

Constraints

Invoked from **InterfaceToProtocolMapping or AssociationToProtocolMapping**

```
\\Select from the transformation input model the operation from a collaboration
let op1=op->select(name=o.name)
P1  \\ Source element is an operation
    o.oclIsTypeOf(Operation) and
P2  \\ The operation is part of an interface
    (o.interface->notEmpty ( ) or
P3   \\ The operation is part of a class
     (o.class -> notEmpty ( ) and
P4   \\Operation is designated as part of a collaboration between classes
     op1->notEmpty() and
P5   \\ Source class is designated as an active class
      ac ->select(name = op1.sourceClass)->notEmpty() and
P6  \\ class is designated as an active class
      ac ->select (name=op1.targetClass)->notEmpty()))
```

PackageMapping (from Semantically Related Element Mapping)

Description

Maps a *SWRadio Spec::Package* from the source model into a *ROSE-RT::Package* in the target model. We consider this mapping to be in the *General UML* mapping group. Packages are defined in the UML specification and are not particular to our case study or the ROSE-RT metamodel.

Context

SWRadio Spec::Package **p**, ROSE-RT::Model **t**

Actions

```
\\ Create a new package with the name of source element as parameter
Package p1= new Package (p.name);
\\ Add package to the target model in the corresponding location
t.add(lookupPathName(p), p1);
```

Constraints

Invoked from **SpecToRoseRTModelMapping**

```
P1  \\ Source element is a Package
    p.oclIsTypeOf(Package)
```

ParameterMapping (from Semantically Related Element Mapping)

Description

Maps a *SWRadio Spec::Parameter* from the source model into a *ROSE-RT::Parameter* in the target model. We consider this mapping to be in the *General UML* mapping group. Parameters are defined in the UML specification and are not particular to our case study or the ROSE-RT metamodel.

Context

SWRadio Spec : :Parameter **p**, ROSE-RT : : Model **t**

Actions

```
\\ Create a new parameter with the name of source element as parameter
Parameter p1= new Parameter (p.name);
\\ Assign the type of the source parameter to the newly created parameter
p1.type.name = p.type.name;
\\ add parameter to the target model
t.add(lookupPathName (p), p1)
\\ invoke other mapping(s)
call ParameterToDependencyMapping;
```

Constraints

Invoked from **OperationMapping**

P1 \\ Source element is a parameter
p.oclIsTypeOf(Parameter)

P2 \\ The owner of the operation is not an interface
not p.parent.parent.oclIsTypeOf(Interface)

ParameterToAttributeMapping (from Interobject Communication Mapping)

Description

Maps a *SWRadio Spec::Parameter* from the source model into a *ROSE-RT::Attribute* in the target model. We consider this mapping to be within the *UML 1.x to ROSE-RT* group of mappings. In our definition, the creation of data classes from an operation definition is done with the purpose of attaching an instance of the data class to a signal. Also in our definition, signals are specific to the ROSE-RT metamodel and thus make it reusable only when considering a ROSE-RT model as target.

Context

SWRadio Spec : :Parameter **p**, ROSE-RT : : Model **t**

Actions

```
\\ Create a new attribute with the name of source element as parameter
Attribute a1= new Attribute (p.name);
\\ Assign the type of the source parameter to the newly created attribute
a1.type.name = p.type.name;
\\ Find the class in which the attribute will be created
Class c1 = t->select (name= p.operation.name+"_DATA_CLASS_FROM_"+p.operation.class.name);
\\ add attribute to the target model
c1.add(a1);
```

Constraints

Invoked from **OperationToDataClassMapping**

P1 \\ Source element is a parameter
p.oclIsTypeOf(Parameter)

ParameterToDependencyMapping (from Interobject Communication Mapping)

Description

Maps a *SWRadio Spec::Parameter::Type* from the source model into a *ROSE-RT::Dependency* in the target model. We consider this mapping to be within the *UML 1.x to ROSE-RT* group of mappings. In ROSE-RT whenever a class is referenced from another class or capsule, it requires that a dependency relationship is defined. The latter make it reusable mostly when considering a ROSE-RT model as target.

Context

SWRadio Spec::Parameter **p**, ROSE-RT : : Model **t**

Actions

```
\\ Create a new dependency relationship
Dependency d1= new Dependency (p.class.name+"_DependsOn_"+p.type.name);
\\ Assign parameter type to supplier role of the Dependency
d1.supplier = p.type.name;
\\ Assign the parent classifier to the client role of the Dependency
d1.client = p.class.name;
\\ Add the new dependency relationship to the target model
t.add(lookupPathName (p.class), d1);
```

Constraints

invoked from **ParameterMapping**

P1 \\ Source element is a parameter
p.oclIsTypeOf(Parameter) **and**

P2 \\ Parameter has a type
p.type->notEmpty ( ) **and**

P2 \\ Dependency does not exist in the source
t->select (name= p.class.name+"_DependsOn_"+p.type.name)->isEmpty

4.1.2 SWRadio Correct implementation of Mappings

In this section we present the verification of the correctness of the application of one of the mappings used in our case study. We present an example of the whole process applied to the GeneralizationMapping to be used to transform the SWRadio PIM. The same process was applied to each mapping definition from section 4.1.1.

4.1.2.1 GeneralizationMapping Correct implementation.

We first recall the GeneralizationMapping Definition from section 4.1.1:

GeneralizationMapping (from Semantically Related Relationship Mapping)

Description

Maps a *SWRadio Spec::Generalization* from the source model into a *ROSE-RT::Generalization* in the target model.

Context

SWRadio Spec::Generalization **g**, Transformation Input::ActiveClass **ac[ ]**, ROSE-RT::Model **t**

Actions

```
\\ create a new generalization
Generalization g1 = new Generalization ( );
\\ Assign the generalized element to the generalization
g1.general= g.general
\\ Find the specialized element in the target model
Classifier c1=t.lookupPathName(g.specific)
\\ Add generalization to the specialized element
c1.add(g1);
```

Constraints

```
        invoked from ClassToDataClassMapping or ClassToCapsuleMapping or InterfaceMapping
P1      \\ Source element is a generalization relationship
        g.oclIsTypeOf(Generalization) and
P2      \\ target metamodel allows only generalizations between similar type of classifiers
        g.specific.type = g.general.type and
P3      \\ classifier with more than one parent
        if (g.specific.generalization.parent->size ( ) > 1 )
P4        \\ classifier is a class
          g.specific.oclIsTypeOf(Class)  and
P5        \\ Classifier is not an active class. ROSE-RT does not allow multiple inheritance on active classes
          ac->select (name= g.specific.name)->isEmpty ( )
        endif
```

We first focus on the constraints defined for the mapping. P1 defines the type of the source element the mapping requires, which in this case, is a generalization relationship. P2 requires both classifiers involved in the generalization to be of the same type. P3 evaluates if the specialized classifier has more than one parent. P4 evaluates if the parent classifier is a Class. Finally, P5 constrains the mapping to be applied only when, in the case of multiple parents, the specialized classifier is not defined as an active class. Based on constraints P1 to P5, we define our initial test model presented in Figure 60.

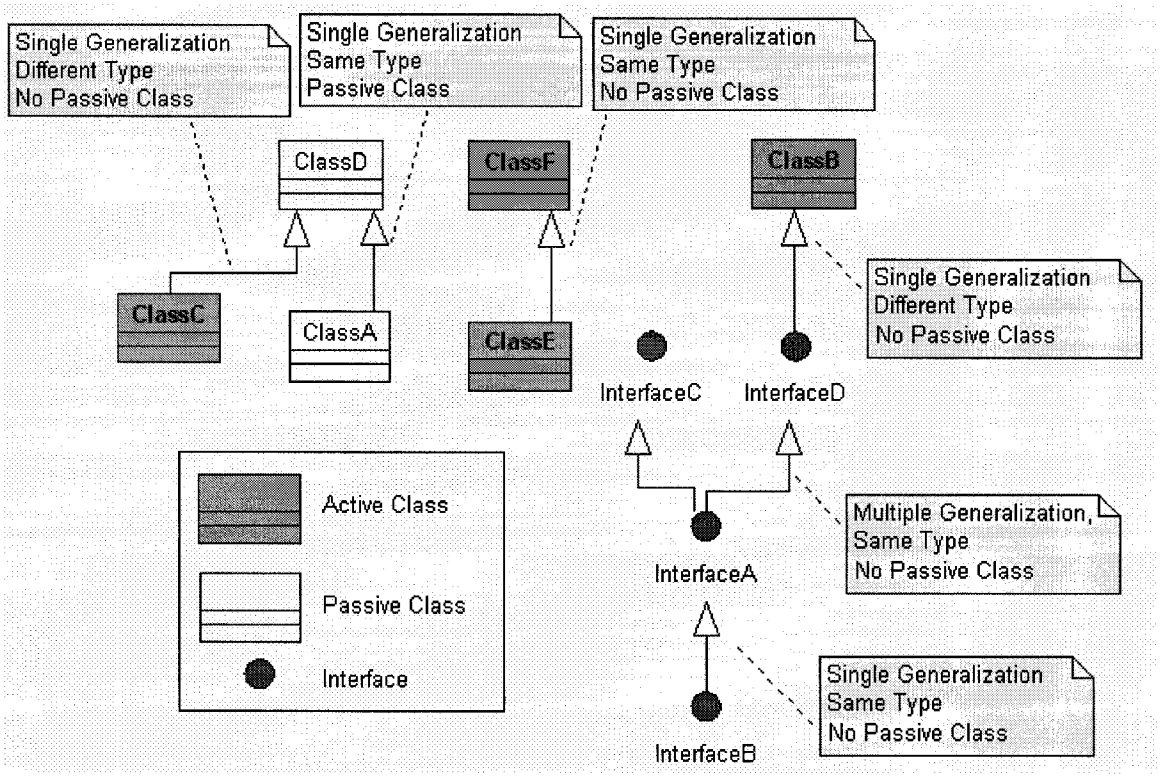


Figure 60. GeneralizationMapping original test model

With the model of Figure 60, we believe that we are covering all the cases imposed by the constraints of the GeneralizationMapping. P1 is covered as we aim at mapping generalization relationships. P2 requires both classifiers being of the same type. We have at least a successful case for P2 (*ClassF* generalizing *ClassE*) and an unsuccessful case in *ClassD* generalizing *ClassC*. P3 looks for a classifier with more than one parent. Most cases are classifiers with a single parent, while InterfaceA that has two parents (*InterfaceC* and *InterfaceD*). P4 looks for parents being a class (*ClassD*, *ClassF* and *ClassB*), and finally P5 looks for parents that are active classes (*ClassF* and *ClassB*). Again, we believe we are covering all cases imposed by the constraints.

Recalling the nature of verifying the correctness of mappings (activity two from our two-step approach) from section 3.1.2, we define a combinational model that covers all valid combinations for GeneralizationMapping. We show the GeneralizationMapping combinational model in Table 29.

Table 29. GeneralizationMapping combinational model

#	P1	P2	P3	P4	P5	Result	Comments
0	0	0	0	0	0	Not Valid	Not a generalization relationship
1	0	0	0	0	1	Not Valid	Not a generalization relationship
2	0	0	0	1	0	Not Valid	Not a generalization relationship
3	0	0	0	1	1	Not Valid	Not a generalization relationship
4	0	0	1	0	0	Not Valid	Not a generalization relationship
5	0	0	1	0	1	Not Valid	Not a generalization relationship
6	0	0	1	1	0	Not Valid	Not a generalization relationship
7	0	0	1	1	1	Not Valid	Not a generalization relationship
8	0	1	0	0	0	Not Valid	Not a generalization relationship
9	0	1	0	0	1	Not Valid	Not a generalization relationship
10	0	1	0	1	0	Not Valid	Not a generalization relationship
11	0	1	0	1	1	Not Valid	Not a generalization relationship
12	0	1	1	0	0	Not Valid	Not a generalization relationship
13	0	1	1	0	1	Not Valid	Not a generalization relationship
14	0	1	1	1	0	Not Valid	Not a generalization relationship
15	0	1	1	1	1	Not Valid	Not a generalization relationship
16	1	0	0	0	0	Not Valid	A classifier cannot be active if it is not a class
17	1	0	0	0	1	0	Target metamodel does not allow for generalization of different types of classifiers
18	1	0	0	1	0	0	Target metamodel does not allow for generalization of different types of classifiers
19	1	0	0	1	1	0	Target metamodel does not allow for generalization of different types of classifiers
20	1	0	1	0	0	Not Valid	A classifier cannot be active if it is not a class
21	1	0	1	0	1	0	Target metamodel does not allow for generalization of different types of classifiers
22	1	0	1	1	0	0	Target metamodel does not allow for generalization of different types of classifiers
23	1	0	1	1	1	0	Target metamodel does not allow for generalization of different types of classifiers
24	1	1	0	0	0	Not Valid	A classifier cannot be active if it is not a class
25	1	1	0	0	1	1	Single Generalization of similar type of classifiers
26	1	1	0	1	0	1	Single Generalization of similar type of classifiers
27	1	1	0	1	1	1	Single Generalization of similar type of classifiers
28	1	1	1	0	0	Not Valid	A classifier cannot be active if it is not a class
29	1	1	1	0	1	0	Target metamodel allows multiple inheritance only in passive classes
30	1	1	1	1	0	0	Target metamodel allows multiple inheritance only in passive classes
31	1	1	1	1	1	1	Multiple Generalization of passive classes

In Table 29 there are 12 combinations that we want to test. Four of them should result in the mapping being executed while the other eight should not. Remember that we are as interested in testing that the mapping will be executed when it should, as in testing the mapping not being executed when it should not.

Based on the original test model from Figure 60 and the combinational model from Table 29, we run our prototype tool. We present our results in Table 30.

Table 30. GeneralizationMapping execution results using an incomplete test model

Element Type	#	Name	P1	P2	P3	P4	P5	Child	Parent	Result
Generalization	17	InterfaceD_To_ClassB	1	0	0	0	1	InterfaceD	ClassB	0
Generalization	18	ClassC_To_ClassD	1	0	0	1	0	ClassC	ClassD	0
Generalization	19	Not in model	1	0	0	1	1			0
Generalization	21	Not in model	1	0	1	0	1			
Generalization	22	Not in model	1	0	1	1	0			
Generalization	23	Not in model	1	0	1	1	1			
Generalization	25	InterfaceB_To_InterfaceA	1	1	0	0	1	InterfaceB	InterfaceA	1
Generalization	26	ClassE_To_ClassF	1	1	0	1	0	ClassE	ClassF	1
Generalization	27	ClassA_To_ClassD	1	1	0	1	1	ClassA	ClassD	1
Generalization	29	InterfaceA_To_InterfaceC	1	1	1	0	1	InterfaceA	InterfaceC	0
Generalization	29	InterfaceA_To_InterfaceD	1	1	1	0	1	InterfaceA	InterfaceD	0
Generalization	30	Not in model	1	1	1	1	0			0
Generalization	31	Not in model	1	1	1	1	1			1

Table 30, we notice that there is no test case for combination number 19, nor test cases for combinations 21, 22, 23, 30 and 31. For the combinations that do have a test case, we verify that the actual results from Table 30 in fact correspond to the expected result from Table 29. The next step is to complete our test model to cover all possible combinations from Table 29. We do so by examining the properties that an element must have to fit into one of the combinations from Table 29 that have not been tested. For example, combination 23 from Table 29 defines an element of type generalization (P1), established between different types of classifiers (not P2), with more than one parent (P3), the child classifier being a class (P4) and the child classifier not being an active class (P5). For example, an element that complies with such description could be defined as a passive class that specializes an interface and an active class. In Figure 61, we present an element (*ClassI*) that fits combination 23 from Table 29 among with other new elements that comply with the constraints defined in the combinations not yet tested.

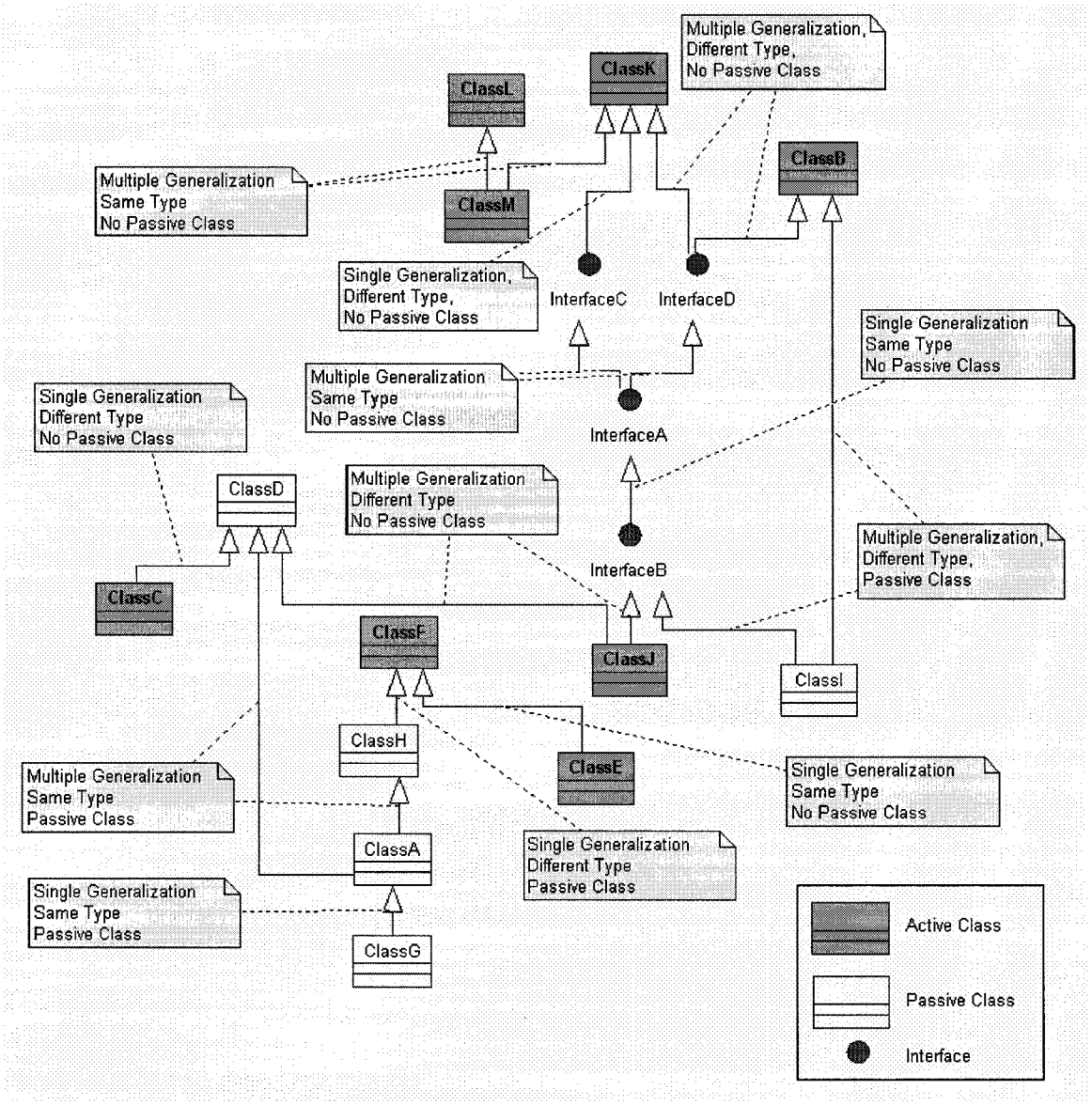


Figure 61. GeneralizationMapping complete test model

Results of the execution of our prototype tool using the new test model from Figure 61 are shown in Table 31.

Table 31. GeneralizationMapping execution results using a complete test model

Element Type	#	Name	P1	P2	P3	P4	P5	Child	Parent	Result
Generalization	17	InterfaceC_To_ClassK	1	0	0	0	1	InterfaceC	ClassK	0
Generalization	18	ClassC_To_ClassD	1	0	0	1	0	ClassC	ClassD	0
Generalization	19	ClassH_To_ClassF	1	0	0	1	1	ClassH	ClassF	0
Generalization	21	InterfaceD_To_ClassB	1	0	1	0	1	InterfaceD	ClassB	0
Generalization	21	InterfaceD_To_ClassK	1	0	1	0	1	InterfaceD	ClassK	0
Generalization	22	ClassJ_To_InterfaceB	1	0	1	1	0	ClassJ	InterfaceB	0
Generalization	22	ClassJ_To_ClassD	1	0	1	1	0	ClassJ	ClassD	0
Generalization	23	ClassI_To_InterfaceB	1	0	1	1	1	ClassI	InterfaceB	0
Generalization	23	ClassI_To_ClassB	1	0	1	1	1	ClassI	ClassB	0
Generalization	25	InterfaceB_To_InterfaceA	1	1	0	0	1	InterfaceB	InterfaceA	1
Generalization	26	ClassE_To_ClassF	1	1	0	1	0	ClassE	ClassF	1
Generalization	27	ClassG_To_ClassA	1	1	0	1	1	ClassG	ClassA	1
Generalization	29	InterfaceA_To_InterfaceC	1	1	1	0	1	InterfaceA	InterfaceC	0
Generalization	29	InterfaceA_To_InterfaceD	1	1	1	0	1	InterfaceA	InterfaceD	0
Generalization	30	ClassM_To_ClassK	1	1	1	1	0	ClassM	ClassK	0
Generalization	30	ClassM_To_ClassL	1	1	1	1	0	ClassM	ClassL	0
Generalization	31	ClassA_To_ClassH	1	1	1	1	1	ClassA	ClassH	1
Generalization	31	ClassA_To_ClassD	1	1	1	1	1	ClassA	ClassD	1

With Table 31 we do verify that our implementation of the GeneralizationMapping does produce results as expected. We also verify that every valid combination is tested by observing that there is at least one element in the test model that fits into each of the valid combinations from Table 29.

As mentioned before, the same process was applied to each mapping definition from section 4.1.1.

4.1.3 SWRadio Complete Definition of Mappings

In this section we present the activities performed to ensure that we have defined a complete set of mappings to transform our SWRadio PIM (*Spec*) into our *tSpec* model to be later used for compliance verification purposes. Recall from section 3.1.3 that the goal of this activity is to verify that every element/relationship from the *Spec* model has been taken into consideration as an originator of an element/relationship of the *tSpec* model. Recall also from 3.1.3 that prototype tools were implemented to help us execute this activity.

Let us continue with the same argument line from the previous section in which we described in detail the efforts we made to ensure that the *GeneralizationMapping* was correctly implemented in our transformation tool. In this section we are to demonstrate that all generalization relationships from the *Spec* model have been considered as originators of elements/relationships of the *tSpec* model.

As mentioned in section 4.1.1, the SWRadio PIM defines 60 generalization relationships between its elements. We run our transformation tool and focus on those generalization relationships. After execution, our transformation tool reports that there are six generalization relationships that have not been used as originators of elements/relationships of the *tSpec* model. Table 32 shows all generalization relationships defined in the SWRadio PIM. The first column shows a sequential number. The second column shows the type of relationship. The third and fourth columns show the SWRadio PIM elements involved in the generalization relationship. Column three presents the General case (parent) in the relationship while column four shows the special case (child). Finally, the fifth column shows the number of times that the relationship was used for mapping purposes. The table considers an ascending order based on the "time used" column.

Table 32. SWRadio Complete definition of Mappings (Generalization example)

#	Relationship Type	General (parent)	Special (child)	Times used
1	Generalization	ComponentRegistration	NamingContext	0
2	Generalization	ComponentReference	NamingContext	0
3	Generalization	PortSupplier	ResourceInt	0
4	Generalization	LifeCycle	ResourceInt	0
5	Generalization	PropertySet	ResourceInt	0
6	Generalization	TestableObject	ResourceInt	0
7	Generalization	SystemException	FileException	1
8	Generalization	SystemException	InvalidFileName	1
9	Generalization	EventType	StateChangeEventType	1
10	Generalization	EventChannel	OutgoingDomainEventChannel	1
11	Generalization	DomainManagementObjectEventType	DomainManagementObjectAddedEventType	1
12	Generalization	DomainManagementObjectEventType	DomainManagementObjectRemovedEventType	1
13	Generalization	EventChannel	IncomingDomainEventChannel	1
14	Generalization	EventType	DomainManagementObjectEventType	1
15	Generalization	EventSupplier	StateChangeEventProducer	1
16	Generalization	EventSupplier	DomainManagementObjectAddedEventProducer	1
17	Generalization	EventSupplier	DomainManagementObjectRemovedEventProducer	1
18	Generalization	EventConsumer	OutgoingDomainEventConsumer	1
19	Generalization	EventConsumer	IncomingDomainEventConsumer	1
20	Generalization	SystemException	IOException	1
21	Generalization	LogStatus	LogConsumer	1

22	Generalization	LogStatus	LogProducer	1
23	Generalization	LogStatus	LogAdministrator	1
24	Generalization	EmbeddedComponent	Resource	1
25	Generalization	EmbeddedComponent	ResourceFactory	1
26	Generalization	Resource	Application	1
27	Generalization	Resource	Device	1
28	Generalization	LoadableDevice	ExecutableDevice	1
29	Generalization	SystemException	InvalidProcess	1
30	Generalization	SystemException	ExecuteFail	1
31	Generalization	Device	LoadableDevice	1
32	Generalization	SystemException	LoadFail	1
33	Generalization	SystemException	RegisterError	1
34	Generalization	SystemException	UnregisterError	1
35	Generalization	SystemException	ApplicationInstallationError	1
36	Generalization	SystemException	ApplicationUninstallationError	1
37	Generalization	Component	EmbeddedComponent	1
38	Generalization	AssemblyDeploymentSpecification	DeviceConfigurationDeploymentSpecification	1
39	Generalization	ComponentPlacement	DCD_ComponentPlacement	1
40	Generalization	PropertyReference	SimpleSequencePropertyReference	1
41	Generalization	PropertyReference	SimplePropertyReference	1
42	Generalization	PropertyReference	StructPropertyReference	1
43	Generalization	PropertyReference	StructSequencePropertyReference	1
44	Generalization	ComponentInstantiationPortReference	ComponentInstantiationLoadedDevicePortReference	1
45	Generalization	ComponentInstantiationPortReference	ComponentInstantiationUsedDevicePortReference	1
46	Generalization	PortReference	ComponentInstantiationPortReference	1
47	Generalization	PortReference	NamingServicePortReference	1
48	Generalization	PortReference	DomainPortReference	1
49	Generalization	DomainFinder	DomainPortReference	1
50	Generalization	ConfigureQueryProperty	SimpleProperty	1
51	Generalization	ConfigureQueryProperty	StructProperty	1
52	Generalization	ConfigureQueryProperty	StructSequenceProperty	1
53	Generalization	Property	TestProperty	1
54	Generalization	ConfigureQueryProperty	SimpleSequenceProperty	1
55	Generalization	Property	ConfigureQueryProperty	1
56	Generalization	SimpleProperty	DeviceArtifactProperty	1
57	Generalization	SimpleProperty	ExecutableProperty	1
58	Generalization	AssemblyDeploymentSpecification	EmbeddedCADSpecification	1
59	Generalization	Partitioning	CADPartitioning	1
60	Generalization	ComponentInstantiation	CADComponentInstantiation	1

From Table 32 we observe that the first six generalization relationships were not mapped into an element/relationship in the *tSpec* model. We also observe that two elements (*NamingContext* and *ResourceInt*) specialize six others (*ComponentRegistration*, *ComponentReference* specialized by *NamingContext*, while *PortSupplier*, *LifeCycle*, *PropertySet* and *TestableObject* are specialized by *ResourceInt*). To gather more information on the generalization relationships not being mapped, we go back to the SWRadio PIM were those elements and relationships are defined (see Figure 62).

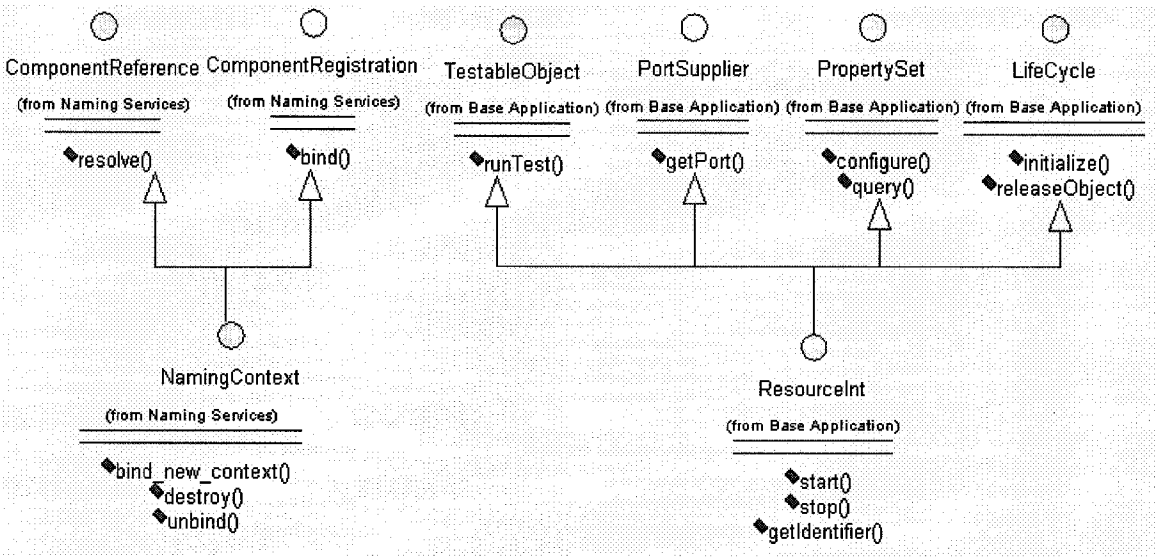


Figure 62. Multiple Generalization of interfaces in the SWRadio PIM

The new information to consider from Figure 62 is that the generalization relationships being discussed are defined between elements of type interface. We then go back to our GeneralizationMapping definition from section 4.1.2 and, using our current elements and relationships, evaluate the mapping constraints to see what the expected output should be. We do so by using our combinational model previously defined in Table 29. Constraint P1 evaluates to true because we are trying to map a generalization relationship. Constraint P2 also evaluates to true because the relationship is established between elements of the same type (interfaces). Constraint P3 also evaluates to true because the child elements (*NamingContext* and *ResourceInt*) specialize more than one parent. From Table 29, we observe that combinations (rows) 29, 30 and 31 are the only valid combinations that consider the first three constraints to evaluate to true. From them, only combination 31 should result in the execution of the mapping (a value of 1 in the expected result column). Such a case requires also constraints P4 and P5 to evaluate to true, the latter requiring that the child element be a passive class. In our present example that is not the case. We are dealing with multiple generalizations of elements of type other than passive classes. We have then found out the reason of why those relationships were not being used for mapping purposes. The next paragraphs discuss our solution, which basically calls for a new mapping to handle the missing relationships.

To define a new mapping, we need first to analyze the problem at hand. Semantics of a generalization relationship define that a specialized (child) element inherits the properties from a general (parent) element. In our case, the parent elements we have are interfaces with one or two operations each. We also have child elements that are also interfaces. What the generalization relationship is trying to accomplish is to 'pass' the properties of the parents to the children. The most common approach to pass such properties is by using generalization relationships but, we repeat, the ROSE-RT metamodel does not allow us to do so in this specific case (multiple generalization of elements other than passive classes). We decide then to define a mapping that will copy such properties into the children as a way to comply with the semantics defined for the generalization relationship. After execution of the new mapping, the result is the addition of operations from the parent interfaces to the child interfaces and the removal of the generalization relationships between the interfaces. We show the result of application of the mapping in Figure 63. Notice how generalization relationships have been removed, and operations defined in the parent interfaces from Figure 62, have been copied into the interfaces of Figure 63 that used to be the child interfaces in Figure 62. We also highlight the title of the figure, which refers to a temporary model. The model in Figure 63 is not the original SWRadio PIM (*Spec*) model, nor the *tSpec* model we are aiming to generate. It is a temporary composition that only represents a single step in the mapping application. Interfaces shown in Figure 63 will be subject to additional mapping applications (i.e., *InterfaceToProtocolMapping*) in the process of generating our *tSpec* model.

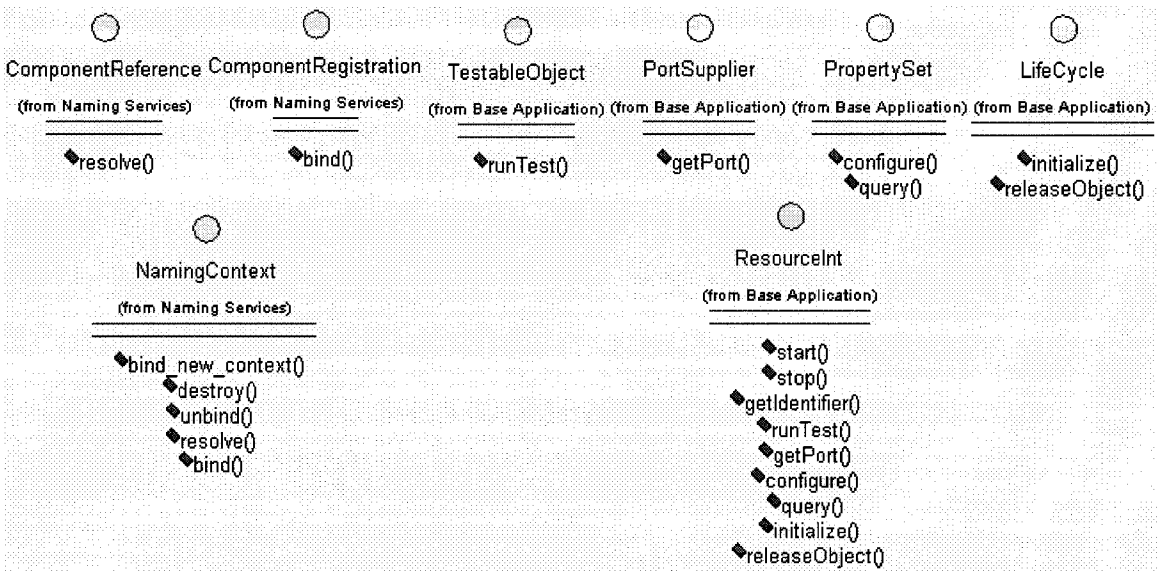


Figure 63. Temporary model after application of the MultipleGeneralizationMapping to the SWRadio PIM

The action part of our new mapping has been illustrated in the previous paragraphs. Now we consider the constraints that will be attached to the new mapping. Such constraints need to consider that the mapping should be applied for generalization relationships, between two interfaces and that the child element inherits from more than one interface. Having those constraints in mind, we define a new mapping as follow:

MultipleGeneralizationMapping (from Semantically Related Relationship Mapping)

Description

Copies the operations of an interface into a child interface when the child interface specializes more than one parent.

Context

SWRadio Spec::Generalization **g**, ROSE-RT::Model **t**

Actions

```
\\ For all operations from the parent interface
for all (g.general.operations) {
\\ Copy operations and properties of the parent interface into the child interface
  g.specific.add(g.general.operation)
}
```

Constraints

invoked from **SpecToRoseRT**

P1 \\ Source element is a generalization relationship
g.oclIsTypeOf(Generalization) **and**

P2 \\ Parent classifier is an interface
g.general.type = interface **and**

P3 \\ Child classifier is an interface
g.specific.type = interface **and**

P4 \\ Child classifier specializes more than one parent
g.specific.generalization.parent->size ( ) > 1

We then repeat step two of our approach that calls for the verification of the correct implementation of the mapping. As the activity 2 has been described at length before (including explicit examples), we decide against including yet another example of it. Full execution of our transformation tool to the ResourceInt interface and its generalization relationships with *TestableObject*, *PortSupplier*, *LifeCycle* and *PropertySet* interfaces from Figure 62, results in the *tSpec* model shown in Figure 64.

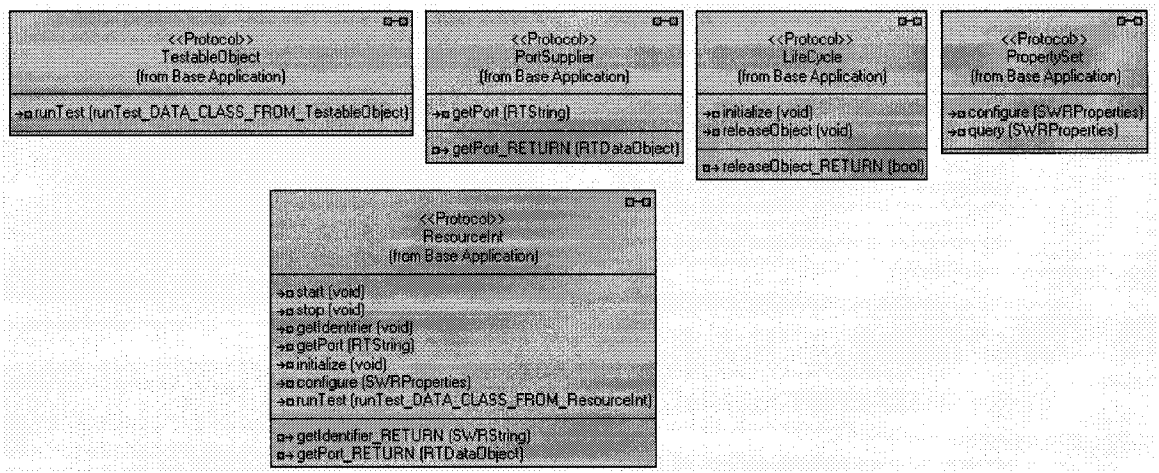


Figure 64. *tSpec* model with protocol class definitions (multiple generalization of interfaces example).

After verifying that the new mapping was implemented correctly, we carry out activity 3 again to verify that we have a complete set of mappings. At this point in time, the reader must have realized that this is an iterative process that requires execution of activities 1, 2 and 3 of our approach until we accomplish our goal: using every element/relationship of the *Spec* model as an originator of elements/relationships of the *tSpec* model. We do so several more times until activity 3 succeeds in mapping every

element/relationship from the SWRadio PIM. Table 33 presents the number of times each element was used during the SWRadio PIM (*Spec*) to *tSpec* transformation.

Table 33. Number of times source elements were used for transformations

Elements used 0 times	0	%
Elements used 1 time	816	94.21
Elements used 2 or more times	48	5.79
Total	866	100 %

From Table 33, we conclude that our goal of using all source elements from the SWRadio PIM at least once in our transformation was accomplished. Table 33 also shows us that 94.21 % of the elements were used only one time during our transformation. 48 elements (which account for 5.79 % of the cases) were used more than once. We checked those cases and found that they correspond to operations in interfaces in which the operations where defined with two or more parameters. The operation was first used to create a signal followed by the creation of a data class to encapsulate the operation's parameters. We conclude this section with Figure 65 that presents a table with all the SWRadio PIM elements that were used two or more times during the *Spec* to *tSpec* transformation.

Element	Type	Owner Element	Type	Owner Element	Type	Owner Element	Type	Owner Element	Type
copy	Operation	FileSystemInt	Interface	File Services	Package	CF Services	Package		
open	Operation	FileSystemInt	Interface	File Services	Package	CF Services	Package		
retrieveById	Operation	LogConsumer	Interface	Lightweight Logging Services	Package	CF Services	Package		
bind	Operation	ComponentRegistration	Class	Naming Services	Package	CF Services	Package		
runTest	Operation	TestableObject	Interface	Base Application	Package				
registerWithEventChannel	Operation	DomainEventChannels	Class	Domain Management Services	Package	CF Control	Package		
unregisterFromEventChannel	Operation	DomainEventChannels	Class	Domain Management Services	Package	CF Control	Package		
registerDevice	Operation	DomainRegistration	Class	Domain Management Services	Package	CF Control	Package		
registerService	Operation	DomainRegistration	Class	Domain Management Services	Package	CF Control	Package		
unregisterService	Operation	DomainRegistration	Class	Domain Management Services	Package	CF Control	Package		
data	Parameter	write	Operation	File	Class	File Services	Package	CF Services	Package
filePointer	Parameter	setFilePointer	Operation	File	Class	File Services	Package	CF Services	Package
data	Parameter	read	Operation	File	Class	File Services	Package	CF Services	Package
length	Parameter	read	Operation	File	Class	File Services	Package	CF Services	Package
mountPoint	Parameter	mount	Operation	FileManager	Class	File Services	Package	CF Services	Package
file_System	Parameter	mount	Operation	FileManager	Class	File Services	Package	CF Services	Package
mountPoint	Parameter	unmount	Operation	FileManager	Class	File Services	Package	CF Services	Package
testId	Parameter	runTest	Operation	UnknownTest	Class	TestableObject	Interface	Base Application	Package
testValues	Parameter	runTest	Operation	UnknownTest	Class	TestableObject	Interface	Base Application	Package
resourceId	Parameter	createResource	Operation	ResourceFactory	Class	Base Application	Package		
qualifiers	Parameter	createResource	Operation	ResourceFactory	Class	Base Application	Package		
resourceId	Parameter	releaseResource	Operation	ResourceFactory	Class	Base Application	Package		
connection	Parameter	connectPort	Operation	PortConnector	Class	Base Application	Package		
connectionId	Parameter	connectPort	Operation	PortConnector	Class	Base Application	Package		
connectionId	*Parameter*	*disconnectPort*	Operation	PortConnector	Class	Base Application	Package		
name	Parameter	create	Operation	ApplicationFactory	Class	Application Services	Package	CF Control	Package
initConfiguration	Parameter	create	Operation	ApplicationFactory	Class	Application Services	Package	CF Control	Package
deviceAssignments	Parameter	create	Operation	ApplicationFactory	Class	Application Services	Package	CF Control	Package
capacities	Parameter	allocateCapacity	Operation	Device	Class	Device Services	Package	CF Control	Package
capacities	Parameter	deallocateCapacity	Operation	Device	Class	Device Services	Package	CF Control	Package
state	Parameter	setAdminState	Operation	Device	Class	Device Services	Package	CF Control	Package
processId	Parameter	terminate	Operation	ExecutableDevice	Class	Device Services	Package	CF Control	Package
name	Parameter	execute	Operation	ExecutableDevice	Class	Device Services	Package	CF Control	Package
options	Parameter	execute	Operation	ExecutableDevice	Class	Device Services	Package	CF Control	Package
parameters	Parameter	execute	Operation	ExecutableDevice	Class	Device Services	Package	CF Control	Package
fs	Parameter	load	Operation	LoadableDevice	Class	Device Services	Package	CF Control	Package
fileName	Parameter	load	Operation	LoadableDevice	Class	Device Services	Package	CF Control	Package
loadKind	Parameter	load	Operation	LoadableDevice	Class	Device Services	Package	CF Control	Package
fileName	*Parameter*	*unload*	*Operation*	*LoadableDevice*	*Class*	*Device Services*	*Package*	*CF Control*	*Package*
associatedDevice	Parameter	addDevice	Operation	AggregateDevice	Class	Device Services	Package	CF Control	Package
associatedDevice	Parameter	removeDevice	Operation	AggregateDevice	Class	Device Services	Package	CF Control	Package
registeringDevice	Parameter	registerDevice	Operation	DeviceManager	Class	Device Services	Package	CF Control	Package
registeredDevice	Parameter	unregisterDevice	Operation	DeviceManager	Class	Device Services	Package	CF Control	Package
registeringService	Parameter	registerService	Operation	DeviceManager	Class	Device Services	Package	CF Control	Package
name	Parameter	registerService	Operation	DeviceManager	Class	Device Services	Package	CF Control	Package
registeredService	Parameter	unregisterService	Operation	DeviceManager	Class	Device Services	Package	CF Control	Package
name	Parameter	unregisterService	Operation	DeviceManager	Class	Device Services	Package	CF Control	Package
componentInstantiationId	Parameter	getComponentImplementationId	Operation	DeviceManager	Class	Device Services	Package	CF Control	Package
releaseObject	Operation	LifeCycle	Interface	Base Application	Package				
bool	Result	releaseObject	Operation	LifeCycle	Interface	Base Application	Package		

Figure 65. Elements applied two or more times in SWRadio PIM transformation

4.1.4 SWRadio Structural Compliance

Before we continue with our case study, we recall from 3.2.1 that we perform structural model compliance by analyzing specification and design models, and reporting their similarities and differences. The aim of finding an exact match between the two models is unrealistic, as many design decisions play a role in defining a design from a specification. Going back to section 3.2.1, recall that our interest lies in the 'only in *Spec*' elements, which are elements that are defined in the *Spec* but not in the *Design*.

In this section we present our case study with respect to the structural compliance of SWRadio designs against the SWRadio PIM. Recall from previous sections that the SWRadio PIM being our specification model has been transformed into a semantically equivalent model (*tSpec*) using that uses the ROSE-RT notation. Through this section we present several *Designs* that may (or may not) be compliant with the *tSpec* model. We use the same rules described in section 3.2.1 to check for structural compliance but with a slight change in the order in which we discuss each of the aspects. We do so to present examples using information already discussed in a previous aspect.

Considering the possibility of future reuse of our structural compliance rules (as we did for mapping definitions), we have included a brief discussion that associates the rule with one of the following groups:

4) *Case Study.* Compliance rules under this group require information that is particular to our case study and its applicability to compliance verification in a different context most probably would imply the modification of the rule to adapt it to the new particular context.
5) *ROSE-RT.* This group of rules might be reused when verifying compliance of models writen in the ROSE-RT language. This type of rules do not require case study specific information.
6) *General.* This group of rules is free of case study specific information, and also independent from the ROSE-RT language.

4.1.4.1 Representation Language

This aspect plays an indirect role in verifying the structural compliance between a *Design* model and a *tSpec* model. Recall from Chapter 3 that this aspect is addressed during the *Spec* to *tSpec* transformation/compliance activities. The outcome of step 1 of our two-step approach for model compliance is a transformed specification (*tSpec*) that uses the same language of the *Design*. In the context of our SWRadio case study, most of this *Spec to tSpec* transformation/compliance with respect of the *Representation Language* aspect has been addressed in section sections 4.1.1 through 4.1.3. In the next paragraphs we address additional input from the SWRadio specification that is needed in order to create the *tSpec* model.

Recall from section 4.1.1 the *ClassToCapsuleMapping* that transforms an active class from the *Spec* model into a capsule in the *tSpec* model. The mapping requires information indicating which of the classes of the *Spec* model are active classes and which ones are not. We do so by providing additional information in the form of an XML document as shown in Figure 66.

```
<?xml version="1.0"?>
<transformationInput>
    <activeClassDefinition>
        <activeClass>CF Control::Device Services::AggregateDevice</activeClass>
        <activeClass>CF Control::Device Services::Device</activeClass>
        <activeClass>CF Control::Device Services::DeviceManager</activeClass>
        <activeClass>CF Control::Device Services::ExecutableDevice</activeClass>
        <activeClass>CF Control::Device Services::LoadableDevice</activeClass>
        <activeClass>CF Control::Device Services::Sequence Diagrams::Comm_User</activeClass>
        <activeClass>CF Control::Device Services::Sequence Diagrams::Node_Boot_Up</activeClass>
        <activeClass>CF Control::Domain Management Services::DomainManager</activeClass>
        <activeClass>Base Application::Resource</activeClass>
        <activeClass>CF Control::Application Services::Application</activeClass>
        <activeClass>SWR Meta-Concepts::EmbeddedComponent</activeClass>
        <activeClass>Base Application::ResourceFactory</activeClass>
        <activeClass>UML::Component</activeClass>
    </activeClassDefinition>
<transformationInput>
```

Figure 66. ActiveClass definition for transformation from *Spec* to *tSpec*

We highlight that within the SWRadio PIM (*Spec* model) there may be many more active classes than the ones we are indicating in Figure 66. Our intention was to work on this SWRadio case study and not to build a complete SWRadio implementation.

4.1.4.2 Interface Realization

Recall from section 3.2.1 that this aspect is defined as a policy decision that impacts the mappings defined to transform the *Spec* model into the *tSpec* model. We chose to implement one port in the *tSpec* for each interface realization in the *Spec*. Our structural comparison will then look for design models implementing individual ports rather than a single port realizing all the interfaces.

This section of the case study first presents an example of the original *Spec* model that includes a *DomainManager* class and interfaces the class realizes. We then present how the *tSpec* looks like after mapping execution. We continue with a *Design* to be verified for compliance and show results of the structural compliance between such a *Design* model and our *tSpec* model.

Figure 67 presents the *Spec* model with the *DomainManager* class and three interfaces. Two interfaces (*ApplicationInstallation* and *DomainEventChannels*) contain two operations each, while the third interface (*DomainHCI*) defines six operations. The figure also shows realization relationships (shown as solid lines) between the *DomainManager* and the interfaces.

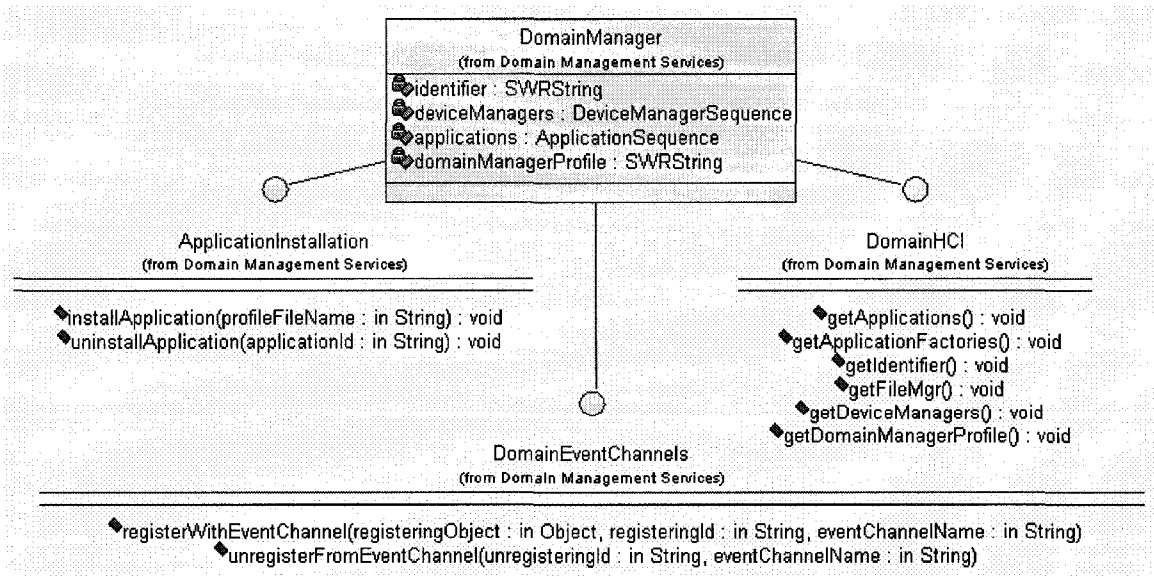


Figure 67. *Spec* DomainManager interfaces

According to our mapping definitions, interfaces from Figure 67 will be transformed into protocol classes in the *tSpec* model. The operations of the interfaces will be transformed into signals in the protocol classes, and the realization relationships from the *DomainManager* to its interfaces will be transformed into composition relationships between the *DomainManager* and the newly created protocol classes. The latter relationships create ports into the *DomainManager* that allow the flow of incoming and outgoing messages in and out the *DomainManager*. Figure 68 presents a class diagram in the *tSpec* model that is semantically equivalent (according to our mapping definitions) to the *Spec* class diagram shown in Figure 67.

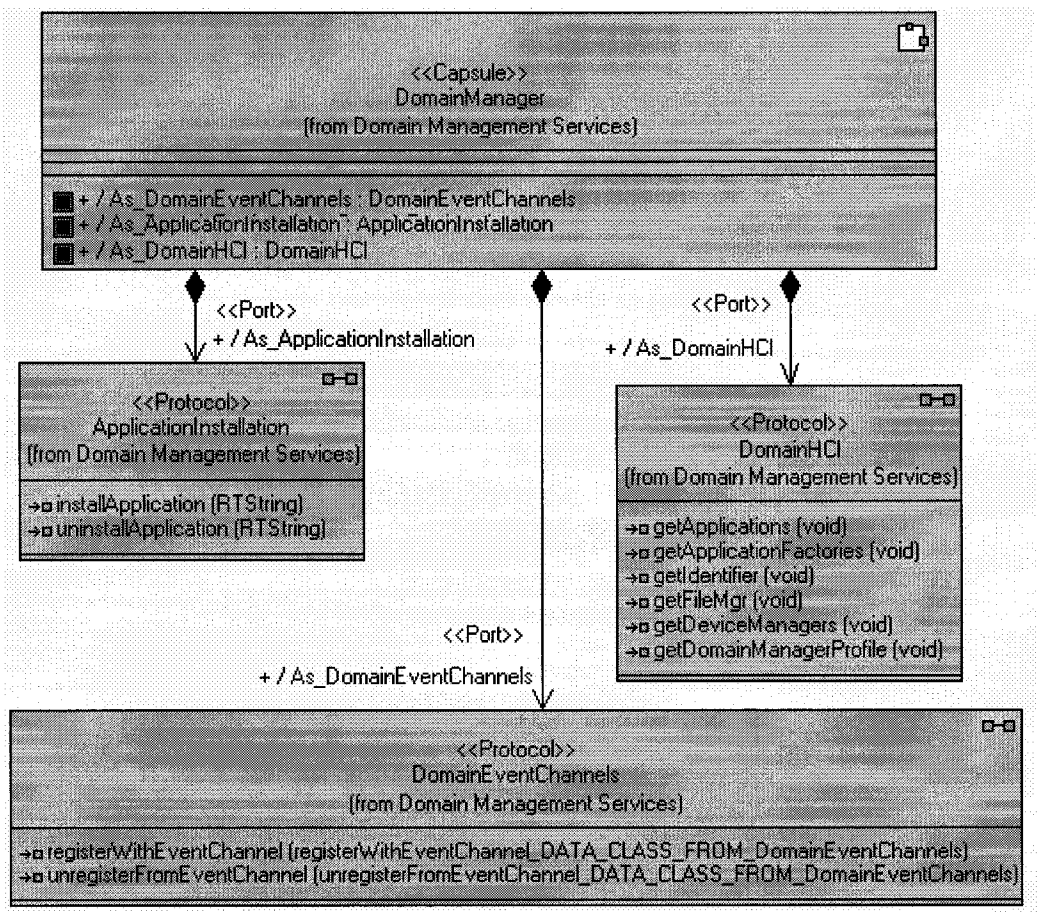


Figure 68. *tSpec* Domain Manager composition relationships to protocol classes

Figure 69 presents the *DomainManager* structure diagram in the *tSpec* model. The diagram shows the communication ports of the *DomainManager*.

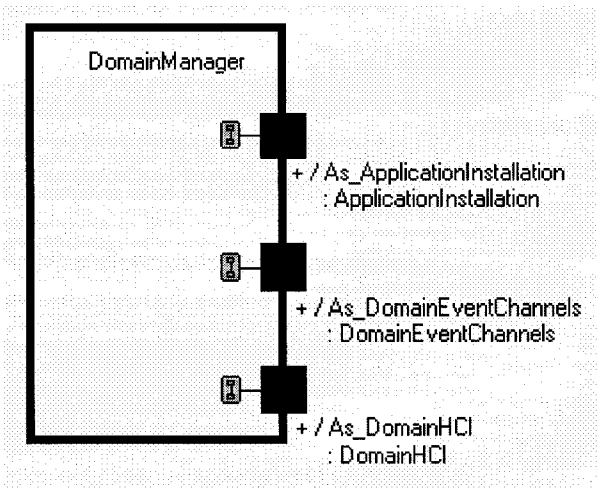


Figure 69. *tSpec* Domain Manager structure diagram with ports

Figure 70 and Figure 71 present a *Design* model to be verified for compliance against the *tSpec* model shown in Figure 68 and Figure 69. The *Design* model presents a single port as opposed to three ports defined in the *tSpec* model. The port is based on the *DomainManagerInterfaces* protocol definition. Although the single port is capable of receiving all the signals of the three ports from the *tSpec* model, our compliance tool will look for the presence of three ports in the *DomainManager*. Under such assumption, the design model should be declared non-compliant with the *tSpec* model.

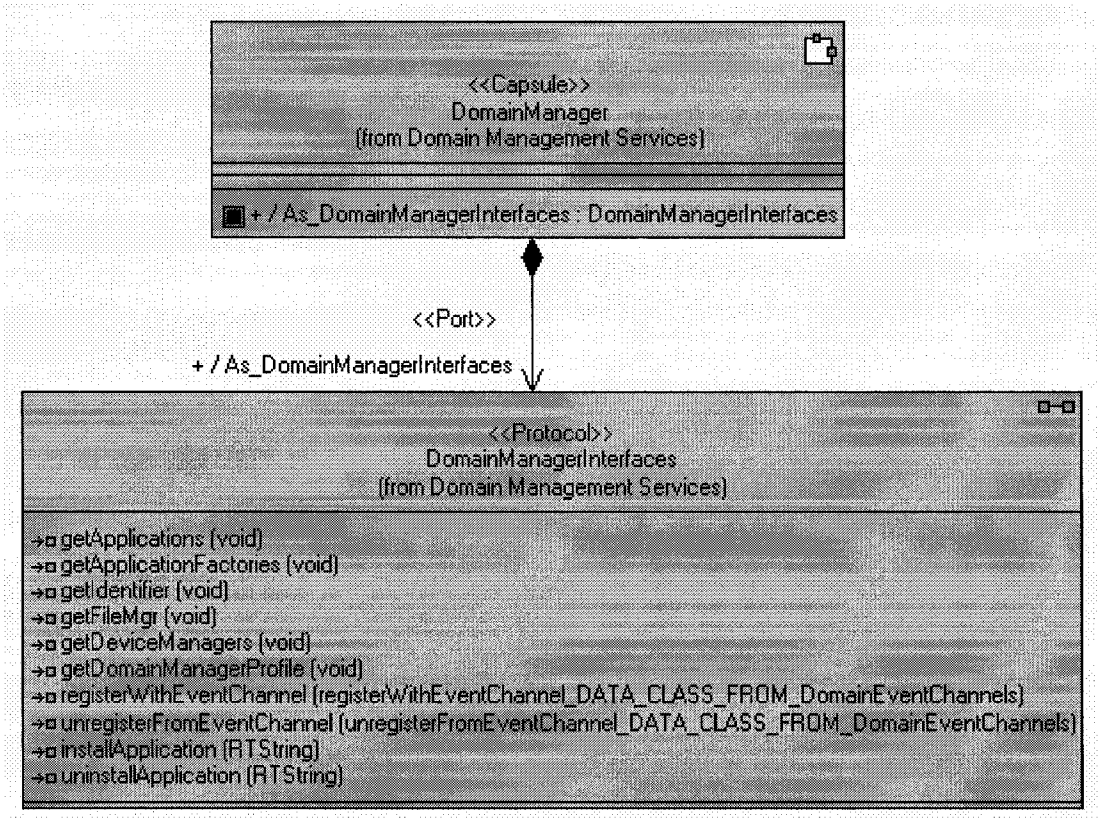


Figure 70. Design DomainManager Ports and Protocols

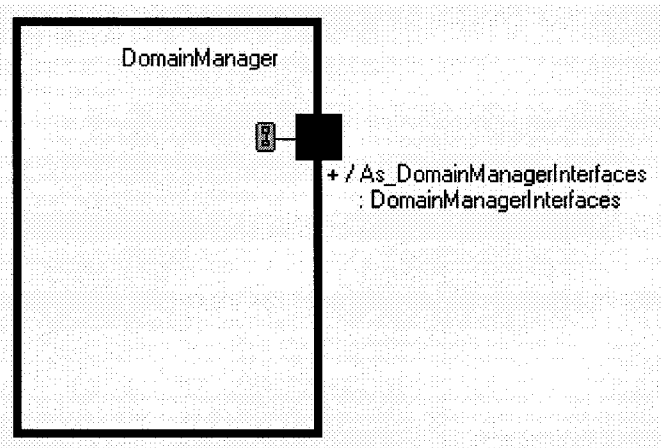


Figure 71. *tSpec* Domain Manager composition relationships to protocol classes

Checking for structural compliance of the *Design* model in Figure 70 and Figure 71 against the *tSpec* model from Figure 68 and Figure 69 produces the results shown in Table 34 through Table 37. Recall from section 3.2.1 that *only in Design* elements and relationships may be originated by design decisions to support the intended functionality

of the system. Although we report the relationship/element from the *Design* that cannot be located in the *tSpec* model, we do not further discuss or consider the information in Table 35 and Table 37. We focus instead on Table 34 and Table 36 that show *only in Spec* elements and relationships. As the tables also show, the missing elements are defined as mandatory elements from the *tSpec*, and thus lead us to declare the *Design* model from Figure 70 and Figure 71 non-compliant with the *tSpec* model from Figure 68 and Figure 69.

Table 34. *tSpec* relationships not found in the Design model

Rel Type	Mandatory/ Optional	Element A	Type	Element B	Type
Composition	MANDATORY	DomainManager	Capsule	As_DomainEventChannels	Port
Composition	MANDATORY	DomainManager	Capsule	As_ApplicationInstallation	Port
Composition	MANDATORY	DomainManager	Capsule	As_DomainHCI	Port

Table 35. Design relationships not found in the *tSpec* model

Rel Type	Element A	Type	Element B	Type
Composition	DomainManager	Capsule	As_DomainManagerInterfaces	Port

Table 36. *tSpec* elements not found in Design model

Element type	Name	Mandatory/ Optional	Owner	Type
Port	As_DomainEventChannels	MANDATORY	DomainManager	Capsule
Port	As_ApplicationInstallation	MANDATORY	DomainManager	Capsule
Port	As_DomainHCI	MANDATORY	DomainManager	Capsule
Protocol	DomainEventChannels	MANDATORY	Domain Management Services	Package
Protocol	ApplicationInstallation	MANDATORY	Domain Management Services	Package
Protocol	DomainHCI	MANDATORY	Domain Management Services	Package

Table 37. Design elements not found in *tSpec* model

Element type	Name	Owner	Type
Port	As_DomainManagerInterfaces	DomainManager	Capsule
Protocol	DomainManagerInterfaces	Domain Management Services	Package

Although the new protocol definition contains all the signals that the missing three protocols define, without additional information we do not relate the single protocol with the missing three. In the *Naming and Role Playing* section below we further discuss this example.

4.1.4.3 Procedure-call-based vs. signal-based communication paradigm

This aspect plays an indirect role in verifying the structural compliance between a *Design* model and a *tSpec* model. Recall from Chapter 3 that this aspect is addressed during the *Spec* to *tSpec* transformation/compliance activities. The outcome of step 1 of our two-step approach for model compliance is a transformed specification (*tSpec*) that includes protocol classes, ports and signals in the *tSpec* model for interface realizations and associations between active classes in the *Spec* model.

In the case of interface realizations, all the information we require is present in the *Spec* model and our transformation tool can map an interface realization into a protocol class, signals and port composition as required. In the case of association between active classes, ROSE-RT defines ports as the only means of communication between capsules. We have already provided the definition of which classes from the *Spec* model are active classes (see the *Representation Language* section above) and which are not. Now there is the need to create a communication infrastructure (that includes protocol classes, signals, ports, and connectors) to allow the interaction between those active classes. At this point we have two options for our implementation. Either we create communication infrastructures for all associations between active classes from the *Spec*, or we provide definition of which associations should be used to create such infrastructure. We further discuss and justify our selection between the two options in the following paragraphs.

Figure 72 illustrates associations between the DomainManager, the DeviceManager and the Device from the SWRadio PIM. In terms of interactions between them, let us use an association between the DeviceManager and the Device to clarify our selected option. In the SWRadio context, the Device registers/unregister itself with the DeviceManager using the registerDevice/unregisterDevice operations from the DomainManager. On the other hand, the Device should not be able to invoke the shutDown operation from the DeviceManager. Given such a condition, attempting to automatically create communication infrastructures based solely on the association that exists between two active classes could lead us to define a protocol class that includes both wanted and unwanted access for the other member of the association. Instead, we decide to

specifically provide additional information stating which associations should be transformed into communication infrastructures. Furthermore, we define which operations within an active class should be transformed into signals in our protocol classes.

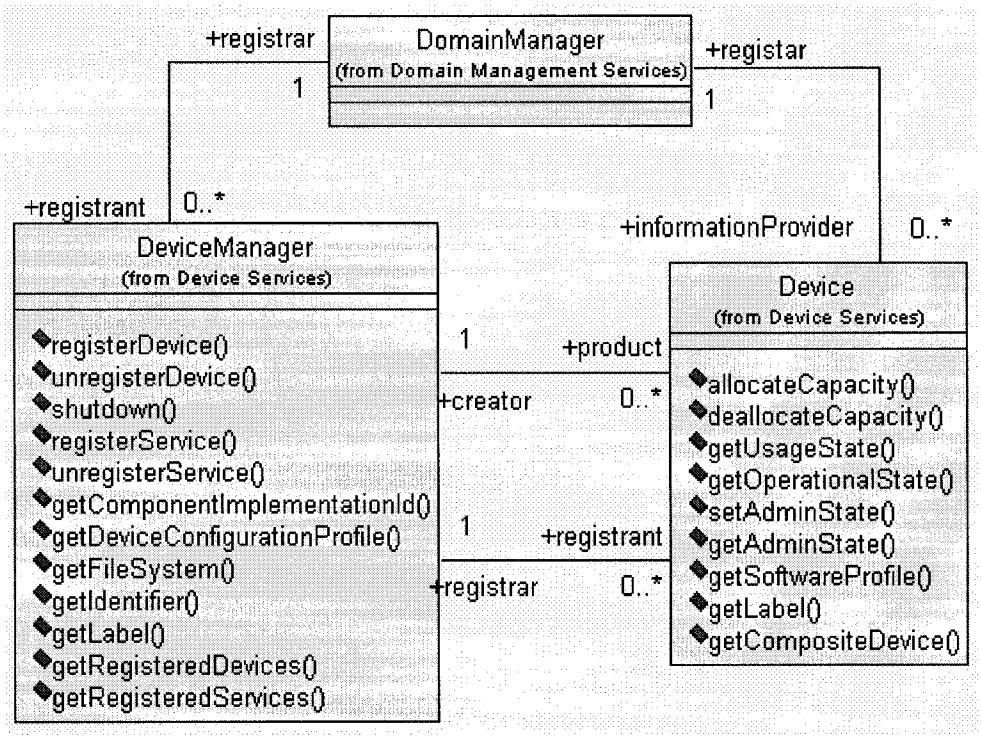


Figure 72. Associations between DomainManager, DeviceManager and Device from the SWRadio PIM

Figure 73 shows a partial definition of the associations (only those related to our current example) from our SWRadio PIM that are to be considered for the creation of communication infrastructures between active classes.

```
<?xml version="1.0"?>
<transformationInput>
        <associationDefinition>
                <association>
                        <associationSource>DomainManager</associationSource>
                        <associationTarget>DeviceManager</associationTarget>
                </association>
                <association>
                        <associationSource>Comm_User</associationSource>
                        <associationTarget>Device</associationTarget>
                </association>
                <association>
                        <associationSource>Device</associationSource>
                        <associationTarget>DeviceManager</associationTarget>
                </association>
        </associationDefinition>
<transformationInput>
```

Figure 73. Association definition for capsule communication.

The result of the definitions on Figure 73 is the creation of specific protocols that will allow communication between the *DomainManager* in the role as *registrar* with the *DeviceManager* using the role of *registrant*, the *Device* in the role of *product* to communicate with the *DeviceManager* in the role of *creator* and finally between the *AggregateDevice* with a *Device*.

We continue by defining which operations between those interactions are to be mapped into signals within the protocol classes. Figure 74 shows a partial definition of those operations. In the figure we define that the *DomainManager* in its role as *registrar* requires accessing the *registerDevice* operation in the *DeviceManager* in its role of *registrant*. The figure also defines operations to be accessed by the *DeviceManager* and the *AggregateDevice* when communicating with the *Device* capsule.

```
<?xml version="1.0"?>
<transformationInput>
        <operationDefinition>
                <operation>
                        <operationName>registerDevice</operationName>
                        <ownerClass>DomainManager</ownerClass>
                        <associationSourceName>DomainManager</associationSourceName>
                        <associationSourceRole>registrar</associationSourceRole>
                        <associationTargetName>DeviceManager</associationTargetName>
                        <associationTargetRole>registrant</associationTargetRole>
                </operation>
                <operation>
                        <operationName>unregisterDevice</operationName>
                        <ownerClass>DeviceManager</ownerClass>
                        <associationSourceName>Device</associationSourceName>
                        <associationSourceRole>product</associationSourceRole>
                        <associationTargetName>DeviceManager</associationTargetName>
                        <associationTargetRole>creator</associationTargetRole>
                </operation>
                <operation>
                        <operationName>removeDevice</operationName>
                        <ownerClass>AggregateDevice</ownerClass>
                        <associationSourceName>Device</associationSourceName>
                        <associationSourceRole></associationSourceRole>
                        <associationTargetName>AggregateDevice</associationTargetName>
                        <associationTargetRole></associationTargetRole>
                </operation>
        </operationDefinition>
<transformationInput>
```

Figure 74. Operation definition for capsule communication.

From Figure 73 and Figure 74 we have the required information we needed to establish a communication mechanism (which include the services that are provided by the communicating elements) between the *DomainManager* and the *DeviceManager*, the *Comm_User* and the *Device*, and the *Device* and *DeviceManager* for our current example.

4.1.4.4 Type correspondence

This aspect plays an indirect role in verifying the structural compliance between a *Design* model and a *tSpec* model. Recall from Chapter 3 that this aspect is addressed during the *Spec* to *tSpec* transformation/compliance activities. The outcome of step 1 of our two-step approach for model compliance is a transformed specification (*tSpec*) that uses the same primitive data types of the *Design*. Figure 75 presents the actual type correspondence information provided in the SWRadio case study for the *Spec to tSpec* transformation. We have established that elements of type *Boolean*, *String*, *UnsignedLongLong*, *UnsignedLong* and *NamingContext* from the *Spec* model will be

defined as elements of type *bool*, *RTString*, *RTDataObject*, *long*, *long* and *RTString* respectively in the *tSpec* model.

```
<?xml version="1.0"?>
<transformationInput>
        <typeCorrespondenceDefinition>
                <typeCorrespondence>
                        <sourceType>Boolean</sourceType>
                        <targetType>bool</targetType>
                </typeCorrespondence>
                <typeCorrespondence>
                        <sourceType>String</sourceType>
                        <targetType>RTString</targetType>
                </typeCorrespondence>
                <typeCorrespondence>
                        <sourceType>Object</sourceType>
                        <targetType>RTDataObject</targetType>
                </typeCorrespondence>
                <typeCorrespondence>
                        <sourceType>UnsignedLongLong</sourceType>
                        <targetType>long</targetType>
                </typeCorrespondence>
                <typeCorrespondence>
                        <sourceType>UnsignedLong</sourceType>
                        <targetType>long</targetType>
                </typeCorrespondence>
                <typeCorrespondence>
                        <sourceType>NamingContext</sourceType>
                        <targetType>RTString</targetType>
                </typeCorrespondence>
        </typeCorrespondenceDefinition>
<transformationInput>
```

Figure 75. Type correspondence definition for transformation from *Spec* to *tSpec*

At this point in the compliance verification process, *Designs* should use the targetTypes from Figure 75 rather than the sourceTypes from the same figure. Our structural compliance tool will look for elements in the Design to be of the exact same types as defined in the *tSpec* model.

4.1.4.5 Naming and Role Playing

Recalling from the example used in the *Interface Realization* section above, we had a single protocol definition in the *Design* model capable to handle all the signals defined in three different protocols in the *tSpec* model. The compliance verification process showed non-compliance as mandatory elements and relationships from the *tSpec* model were not found in the *Design* model (see *Interface Realization* above). In this section we provide a solution to this challenge. By providing additional information, we can relate elements of

the *Design* that are playing the role of an element in the *tSpec* when they use different names. Figure 76 presents an XML document where we establish that *DomainManagerInterfaces* in the *Design* plays the role of *DomainEventChannels* in the *tSpec*. Please notice that, in this section, we are not establishing a link between the *DomainManagerInterfaces* and the other two protocol classes used in the *Interface Realization* section above. We will do so in the *Multi-Role* section later in this chapter.

```
<?xml version="1.0"?>
<nameEquivalenceDefinition>
        <nameEquivalence>
                <nameInSpec>DomainEventChannels</name>
                <nameInDesign>DomainManagerInterfaces</type>
        </nameEquivalence>
</nameEquivalenceDefinition>
```

Figure 76. Name equivalence definition for elements from the Design playing a role of elements in the *tSpec* model

Executing structural compliance with this new information, the output of the process is similar to the one presented in Table 34 and Table 35 from the *Interface Realization* section above, but differs in the results for Table 36 and Table 37. The new results are shown in Table 38 and Table 39. Without changing the *Design* model, this time the *DomainEventChannels* protocol class from the *tSpec* was linked to the *DomainManagerInterfaces* protocol class in the *Design*. The tool then found that there is an element in the *Design* (*DomainManagerInterfaces*) that plays the role of the *DomainEventChannels* element in the *tSpec* and thus does not report it as an *only in Spec* element. A similar situation takes place for *only in Design* elements.

Table 38. *tSpec* elements not found in Design model

Element type	Name	Mandatory/ Optional	Owner	Type
Port	As_DomainEventChannels	MANDATORY	DomainManager	Capsule
Port	As_ApplicationInstallation	MANDATORY	DomainManager	Capsule
Port	As_DomainHCI	MANDATORY	DomainManager	Capsule
Protocol	ApplicationInstallation	MANDATORY	Domain Management Services	Package
Protocol	DomainHCI	MANDATORY	Domain Management Services	Package

Table 39. Design elements not found in *tSpec* model

Element type	Name	Owner	Type
Port	As_DomainManagerInterfaces	DomainManager	Capsule

The output of the model compliance verification process is negative as there are still mandatory elements and relationships from the *tSpec* model that were not found in the *Design* model (*only in Spec* elements and relationships). We further discuss this example in the *Multi-role Players* section below.

4.1.4.6 Mandatory vs. Optional elements

For this section of the case study, we use the same *tSpec* and *Design* model used in the *Interface Realization* section above (see Figure 68 and Figure 69 for the *tSpec* model and Figure 70 and Figure 71 for the *Design* model). Recalling our previous compliance verification results, the *Design* was considered non-compliant because mandatory elements from the *tSpec* model were not found in the *Design* model. For this section we use an XML model (shown in Figure 77) that labels elements from the *tSpec* model as *optional* elements. For this example we are defining the *DomainEventChannels* and *ApplicationInstallation* protocol classes as optional elements from the *tSpec* model. The same situation applies to the *As_DomainEventChannels* and *As_ApplicationInstallation* ports also from the *tSpec* model.

```
<?xml version="1.0"?>
<optionalInputDefinition>
        <optionalElement>
                <name>DomainEventChannels</name>
                <type>Protocol</type>
        </optionalElement>
        <optionalElement>
                <name>ApplicationInstallation</name>
                <type>Protocol</type>
        </optionalElement>
        <optionalElement>
                <name>As_DomainEventChannels</name>
                <type>Port</type>
        </optionalElement>
        <optionalElement>
                <name>As_ApplicationInstallation</name>
                <type>Port</type>
        </optionalElement>
</optionalInputDefinition>
```

Figure 77. Optional definition for elements from the *tSpec* model

We execute our compliance verification tool using the *tSpec* model from Figure 68 and Figure 69, the *Design* model from Figure 70 and Figure 71 and the *optional* element definition from Figure 77. The results are the same for *only in Design* elements (see Table 35 and Table 37) from those shown *Interface Realization* section above, but differ in the results for Table 34 and Table 36 that present *only in Spec* elements and relationships. We show the new results in Table 40 and Table 41. Notice how Table 40 and Table 41 differ from Table 34 and Table 36 in the MANDATORY/OPTIONAL labeling of *tSpec* elements. We report the missing elements and label the as *optional.* With this new information, the presence of missing elements from the *tSpec* model will not label a design model as non-compliant. We consider important, though, to report it because this information may be used for other purposes.

Table 40. *tSpec* relationships not found in the Design model

Rel Type	Mandatory/ Optional	Element A	Type	Element B	Type
Composition	OPTIONAL	DomainManager	Capsule	As_DomainEventChannels	Port
Composition	OPTIONAL	DomainManager	Capsule	As_ApplicationInstallation	Port
Composition	MANDATORY	DomainManager	Capsule	As_DomainHCI	Port

Table 41. *tSpec* elements not found in Design model

Element type	Name	Mandatory/ Optional	Owner	Type
Port	As_DomainEventChannels	OPTIONAL	DomainManager	Capsule
Port	As_ApplicationInstallation	OPTIONAL	DomainManager	Capsule
Port	As_DomainHCI	MANDATORY	DomainManager	Capsule
Protocol	DomainEventChannels	OPTIONAL	Domain Management Services	Package
Protocol	ApplicationInstallation	OPTIONAL	Domain Management Services	Package
Protocol	DomainHCI	MANDATORY	Domain Management Services	Package

After execution of the compliance verification process, we found the *Design* model to be non-compliant as there are still mandatory elements from the *tSpec* model (protocol *DomainHCI* and port *As_DomainHCI*) that were not found in the *Design* model.

4.1.4.7 Designs implementing only a subset of a specification

We recall from section 3.2.1 that this aspect allows *design implementers* to target specific portions of the specification model, while intentionally not implementing some others. We emphasize that enabling this option helps the *design implementers* in

verifying partial models, but should not change the compliant (or not) outcome of the verification process. Take for example the *tSpec* model from Figure 68 and Figure 69, and the *Design* model from Figure 70 and Figure 71. If the *design implementers* are interested verifying its partial work with respect of the *DomainEventChannels* and *ApplicationInstallation* protocols and its realizations (ports), they can add such information in the form of the XML document shown in Figure 78 below:

```
<?xml version="1.0"?>
<targetElementDefinition>
        <targetElement>
                <name>DomainEventChannels</name>
                <type>Protocol</type>
        </targetElement >
        <targetElement >
                <name>ApplicationInstallation</name>
                <type>Protocol</type>
        </targetElement >
        <targetElement >
                <name>As_DomainEventChannels</name>
                <type>Port</type>
        </targetElement >
        <targetElement >
                <name>As_ApplicationInstallation</name>
                <type>Port</type>
        </targetElement >
</ targetElement Definition>
```

Figure 78. Optional definition for elements from the *tSpec* model

With no additional information (i.e., role playing or optional elements) the outcome of the compliance verification process is shown in Table 42 and Table 43. At first glance, the output shows the same results than in the *Interface Realization* section, with the difference of some elements being labeled as *TARGET* elements. The *Design* is still non-compliant because there are three mandatory relationships and six elements from the *tSpec* that were not found in the *Design* model. The key difference between the two outputs (the one from the *Interface Realization* section and the one showed in Table 42 and Table 43) is that the *design implementers* are only two relationships and four elements shy of reaching their goal. In other words, their intent is to implement *DomainEventChannels* and *ApplicationInstallation* protocol classes and the *As_DomainEventChannels* and *As_ApplicationInstallation* ports because that is the target they are setting for their *Design*.

Table 42. *tSpec* relationships not found in the Design model

Rel Type	Mandatory/ Optional	Element A	Type	Element B	Type
Composition	MANDATORY TARGET	DomainManager	Capsule	As_DomainEventChannels	Port
Composition	MANDATORY TARGET	DomainManager	Capsule	As_ApplicationInstallation	Port
Composition	MANDATORY	DomainManager	Capsule	As_DomainHCI	Port

Table 43. *tSpec* elements not found in Design model

Element type	Name	Mandatory/ Optional	Owner	Type
Port	As_DomainEventChannels	MANDATORY TARGET	DomainManager	Capsule
Port	As_ApplicationInstallation	MANDATORY TARGET	DomainManager	Capsule
Port	As_DomainHCI	MANDATORY	DomainManager	Capsule
Protocol	DomainEventChannels	MANDATORY TARGET	Domain Management Services	Package
Protocol	ApplicationInstallation	MANDATORY TARGET	Domain Management Services	Package
Protocol	DomainHCI	MANDATORY	Domain Management Services	Package

We emphasize that even when the *design implementers* implement the *DomainEventChannels* and *ApplicationInstallation* protocol classes and the *As_DomainEventChannels* and *As_ApplicationInstallation* ports, the *Design* will still be non-compliant because it will be missing the *DomainHCI* protocol and *As_DomainHCI* port. Still, the *design implementers* can consider their goals met when a table like the one shown in Table 42 and Table 43 includes only mandatory elements and relationships that have not been indicated as *targets* of the *Design* model.

4.1.4.8 Package Contents

We recall that this challenge implies having (or not) the same package structure in the *Design* model as defined in the *tSpec* model. For our SWRadio case study we followed the same policy we defined before in section 3.2.1. We consider unnecessary to add burden on the *design implementers* by forcing them to have the same package structure in their design models as defined in the specification model. Our structural compliance tool does not consider the package location of a *Design* element to give a found or not found result. We do not further discuss this issue.

4.1.4.9 Multi-role players

Recalling from the example used in the *Interface Realization* section above, we have a single protocol definition in the *Design* model capable to handle all the signals defined in three different protocols in the *tSpec* model. The compliance verification process showed non-compliance as mandatory elements and relationships from the *tSpec* model were not found in the *Design* model (see *Interface Realization* above). In this section we provide a solution to this challenge. By providing additional information, we relate elements of the design that are playing the role of an element in the *tSpec* when they use different names. Figure 79 presents an XML description where we define that *DomainManagerInterfaces* protocol class in the *Design* plays the role of the *DomainEventChannels,* the *ApplicationInstallation* and the *DomainHCI* protocol classes from the *tSpec* model. We did the same for the port *As_DomainManagerInterfaces* in the *Design* playing the role of the *As_DomainEventChannels,* the *As_ApplicationInstallation* and the *As_DomainHCI* ports in the *tSpec.*

```
<?xml version="1.0"?>
<nameEquivalenceDefinition>
        <nameEquivalence>
                <nameInSpec>DomainEventChannels</name>
                <nameInDesign>DomainManagerInterfaces</type>
        </nameEquivalence>
        <nameEquivalence>
                <nameInSpec>ApplicationInstallation</name>
                <nameInDesign>DomainManagerInterfaces</type>
        </nameEquivalence>
        <nameEquivalence>
                <nameInSpec>DomainHCI</name>
                <nameInDesign>DomainManagerInterfaces</type>
        </nameEquivalence>
        <nameEquivalence>
                <nameInSpec>As_DomainEventChannels</name>
                <nameInDesign>As_DomainManagerInterfaces</type>
        </nameEquivalence>
        <nameEquivalence>
                <nameInSpec>As_ApplicationInstallation</name>
                <nameInDesign>As_DomainManagerInterfaces</type>
        </nameEquivalence>
        <nameEquivalence>
                <nameInSpec>As_DomainHCI</name>
                <nameInDesign>As_DomainManagerInterfaces</type>
        </nameEquivalence>
</nameEquivalenceDefinition>
```

Figure 79. Name equivalence definition for elements from the Design playing a role of elements in the *tSpec* model

Executing structural compliance with this new information, the output of the process is that the *Design* is compliant with the *tSpec* model, as all the elements and relationships defined in the *tSpec* were located in the *Design* model. When regenerated, similar tables as the ones shown in previous sections came up empty (again, all elements were located in both models) and thus we do not reproduce them.

4.1.4.10 Design implementing more than one specification

We recall from section 3.2.1 that a design may contain elements that are defined in a different specification that the one it is being used for compliance verification purposes. We postulate that those elements do not constitute a basis to rule non-compliance. Instead, we have the tools supporting our proposed compliance verification process simply report such elements to the user. In our SWRadio case study we do not attempt to model a second specification, and thus provide no examples for this challenge. This is a topic that will be left for future work related to our research.

4.1.5 SWRadio Behavioral Compliance

This section presents the application of the two-step approach for compliance verification focusing on the behavioral compliance of a SWRadio design model. The section is composed of several test cases in which we focus on showing how the detection of non-compliant points was achieved. For each test case, we present a variation of a design model that was built incrementally. In other words, test case $n+j$ uses design elements introduced in test cases $n+j-1$, $n+j-2$, ..., n, unless the test case itself defines otherwise.

Before we proceed with this section, we would like to emphasize that the examples presented in this section are academic in purpose and do not fully show the complexity of the *Design* models used. The Design models used contain more than 120 class definitions and each of them was successfully compiled for some of the examples in this section. We invite the reader to consult Appendix A to look at a log report about the compilation of one of our design models. The Appendix shows the classes that were generated and further compiled.

Test cases were chosen to cover at least one of each of the causes for non-compliance described in section 3.2.2. We developed test cases to show the following non-compliant causes:

- Non-executable model
- Incorrect sending of messages
- Required port not defined
- Required connector not defined
- missing or incorrect data attached to a message
- multi scenario execution

We also tried our best to avoid duplication of examples. We judged that duplication would only lead us to more figures and bigger tables, but not much in terms of added value for the case study itself.

Table 44 presents a summary of the test cases discussed in this section. The first column indicates the test number. The second column provides a brief description on a particular condition of the SWRadio design model. The third column shows the failure/success of the test execution. Finally, the fourth column presents the type of non-compliant point found in cases where the test execution resulted in a failure.

Table 44. SWRadio behavioral compliance test cases

Test	Design model description	Result	Non-compliant point
1	Not an executable design	failure	Non-executable model
2	No messages being sent	failure	Incorrect sending of messages
3	Missing a port	failure	Required port not defined
4	Missing a connector	failure	Required connector not defined
5	Design does not send three messages	failure	Incorrect sending of messages
6	Design does not send two messages	failure	Incorrect sending of messages
7	Design missing data to be attached to a message	failure	missing or incorrect data attached to a message
8	Design with incompatible data to be attached to a message	failure	missing or incorrect data attached to a message
9	Design does not send one message	failure	Incorrect sending of messages
10	Ready to execute ReleaseCompositeDevice scenario	success	
11	Not ready for releaseAggregatedDevice scenario	failure	Incorrect sending of messages
12	Ready to execute releaseAggregatedDevice scenario	success	
13	Not ready to execute releaseAggregatedDevice scenario immediately after ReleaseCompositeDevice scenario.	failure	multi scenario execution
14	Ready to execute releaseAggregatedDevice scenario immediately after ReleaseCompositeDevice scenario	success	

In this behavioral compliance section of the case study, the first set of test cases focus on testing a SWRadio design model for its capabilities to execute independent scenarios as defined by the *releaseCompositeDevice* and the *releaseAggregatedDevice* sequence diagrams shown in Figure 80 and Figure 81 from [80]. We then test the *releaseAggregatedDevice* immediately after the *releaseCompositeDevice* scenario creating an inter-scenario dependency between the two scenarios.

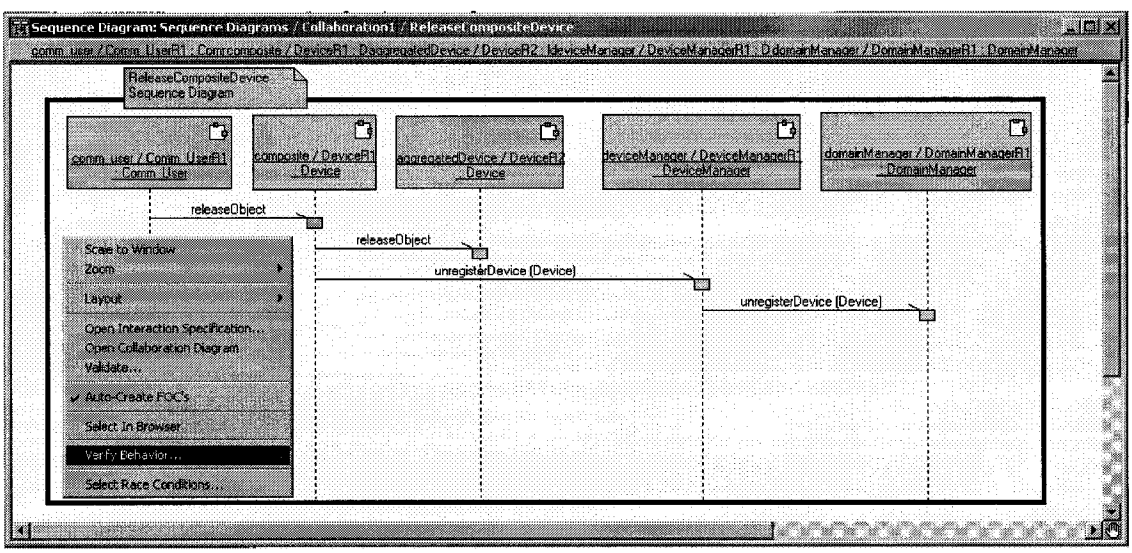


Figure 80. ReleaseCompositeDevice specification sequence diagram

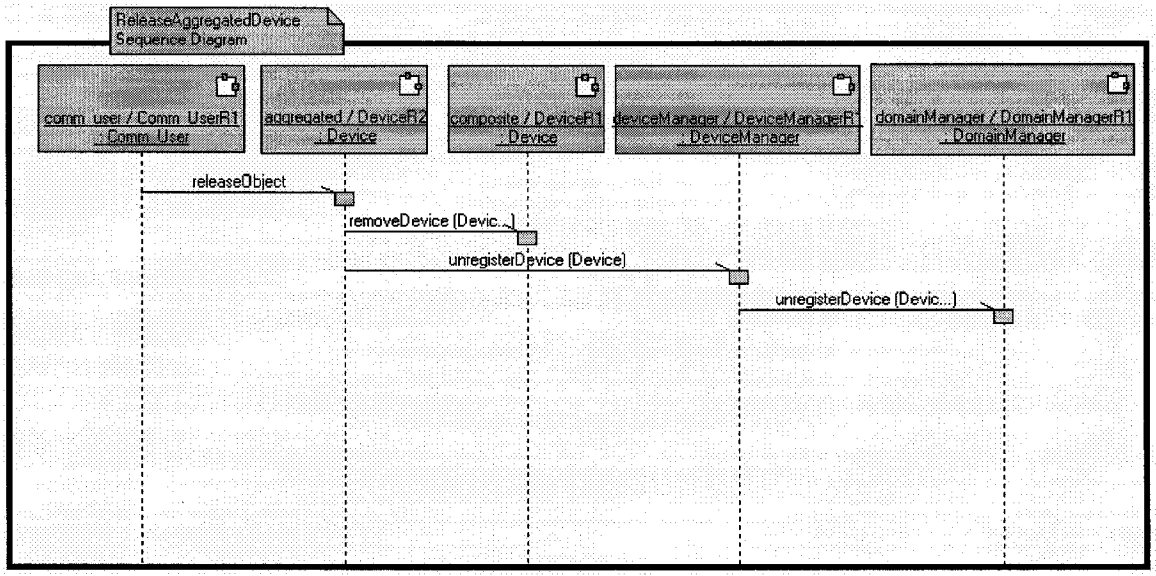


Figure 81. ReleaseAggregatedDevice specification sequence diagram

The SWRadio design we used for testing was based on the structure diagram presented in Figure 82. The figure illustrates collaboration between entities involved in the sequence diagrams from Figure 80 and Figure 81. We highlight that the *DeviceR1* and *DeviceR2* roles of the *Device* capsule in our SWRadio Design, play the roles of the *composite* and *aggregate* instances used in the sequence diagrams from Figure 80 and Figure 81. Roles of SWRadio entities are denoted by boxes with the name of the role followed by a colon and the name of the SWRadio entity. Small square boxes on the boundaries of the SWRadio roles illustrate ports. The name of the ports appears besides the port icon followed by the name of the protocol the port is considered an instance of. A line connecting ports denotes connectors.

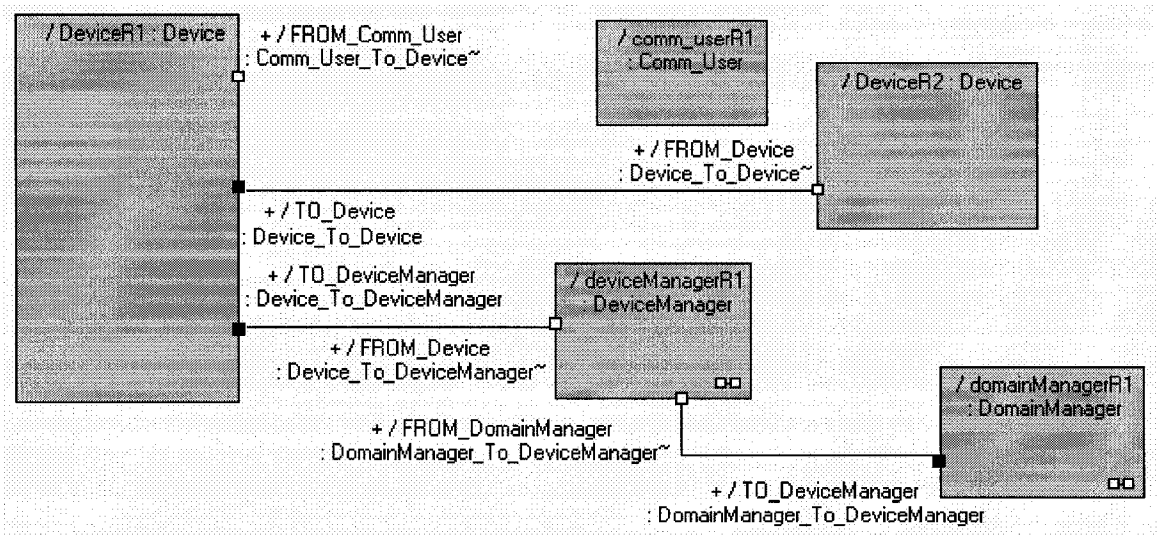


Figure 82. SWRadio Design structure diagram

The next subsections describe our test cases as outlined before: first we focus on testing a SWRadio design model for its capabilities to execute independent scenarios as defined by the *releaseCompositeDevice* and the *releaseAggregatedDevice* sequence diagrams shown in Figure 80 and Figure 81 from [80]. We then test the *releaseAggregatedDevice* immediately after the *releaseCompositeDevice* scenario creating an inter-scenario dependency between the two scenarios.

ReleaseCompositeDevice sequence diagram compliance verification.

Test cases 1-10 attempt to verify that a SWRadio design is capable of executing the same sequence of message interchanges as defined in the sequence diagram of Figure 80.

Test 1: Design model is not an executable design. The first design model we test does not implement behavior at all, and produces compiler errors that prevent the executability of the model. The model fails to compile due to missing C++ code. Figure 83 presents common errors received by the compiler. The full message log produced in this attempt is shown in Appendix A.

```
../src/DeviceManager.cpp(320) : error C4716: 'DeviceManager_Actor::getComponentImplementationId' : must return a value
../src/DeviceManager.cpp(328) : error C4716: 'DeviceManager_Actor::getDeviceConfigurationProfile' : must return a value
../src/DeviceManager.cpp(336) : error C4716: 'DeviceManager_Actor::getFileSystem' : must return a value
../src/DeviceManager.cpp(344) : error C4716: 'DeviceManager_Actor::getIdentifier' : must return a value
../src/DeviceManager.cpp(352) : error C4716: 'DeviceManager_Actor::getLabel' : must return a value
../src/DeviceManager.cpp(360) : error C4716: 'DeviceManager_Actor::getRegisteredDevices' : must return a value
../src/DeviceManager.cpp(368) : error C4716: 'DeviceManager_Actor::getRegisteredServices' : must return a value
```

Figure 83. Compile errors: operation's return value missing

The errors mainly relate to operations failing to return a value as defined in their respective signatures. Although fixing those errors may seem not related to our compliance verification approach, one of the premises that we have defined for our process is that we need to be able to execute our models in order to verify their behavior. So there is an implicit need to fix those errors in order to verify for behavioral compliance.

Test 2: Design model with no messages being sent/received. This time, although an executable SWRadio Design, the model does not define state machines for the entities involved in our test case. The result of the RQA verification process is shown in Figure 84. Test errors of the same verification attempt are shown in Figure 85.

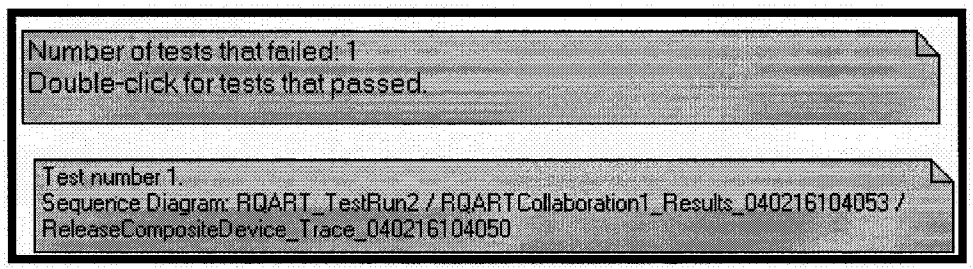


Figure 84. ReleaseCompositeDevice test result: failure. No state machines defined

```
10:40:54| Differences between specification: "ReleaseCompositeDevice" and trace:
"ReleaseCompositeDevice_Trace_040216104050":
10:40:54| Warning:   Specification:   " : releaseObject" was unexpected
10:40:54| Warning:   Specification:   " : releaseObject" was unexpected
10:40:54| Warning:   Specification:   " : unregisterDevice" was unexpected
    10:40:54| Warning:   Specification:   " : unregisterDevice" was unexpected
```

Figure 85. ReleaseCompositeDevice test errors: no messages sent/received

The results of the test indicate non-compliant points due to *incorrect sending of messages*. The reason being that, as the design model contains no state machines, the execution was not capable to send or receive any messages. While the specification

defines four messages to be sent and received, the execution was not capable to produce them. Figure 86 shows the messages received by the instances that play a role as defined in Figure 80. None of them are related to the messages from Figure 80, as all the messages found in Figure 86 correspond to system messages to create, initialize and further destroy the instances that play a role in the releaseCompositeDevice scenario execution.

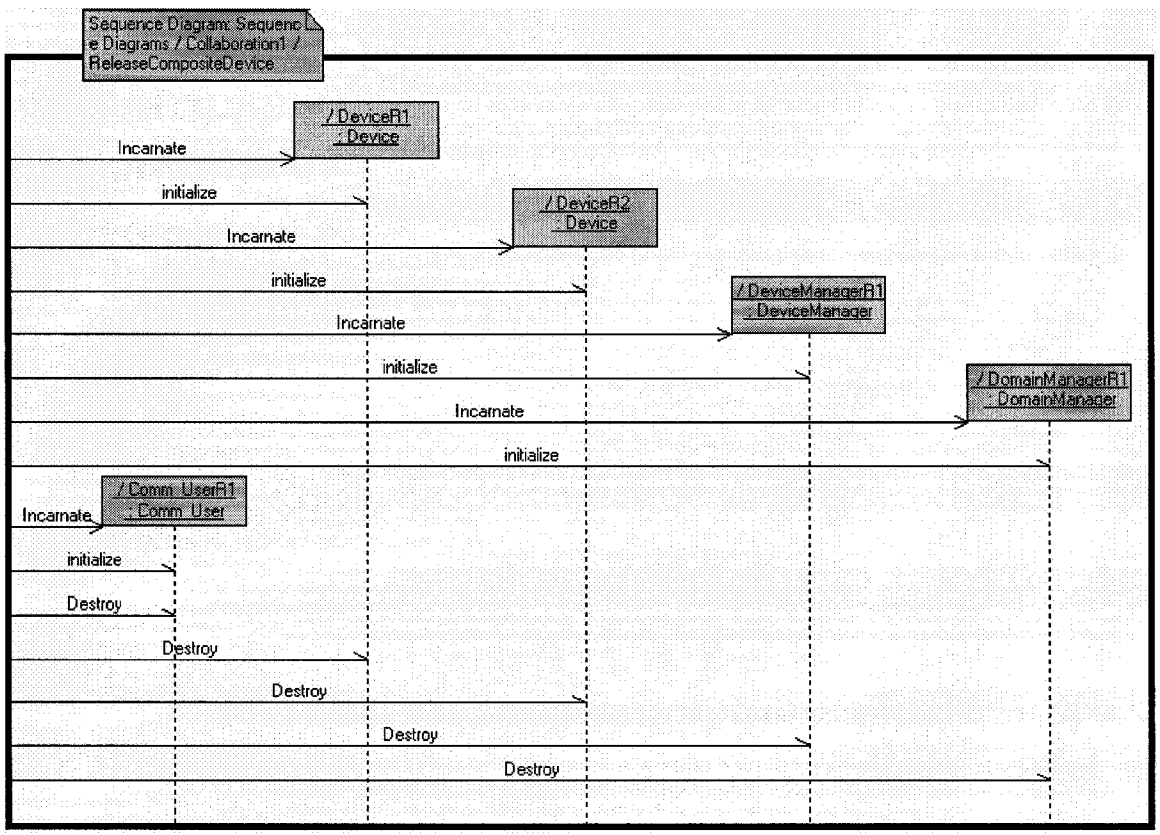


Figure 86. ReleaseCompositeDevice execution sequence diagram. No messages sent/received

Test 3: Design model missing a port. This test is based on the state machine definition for the *Comm_User* capsule presented in Figure 87. The figure shows on the left side a browser with the elements that constitute the *Comm_User* capsule. On the center of the figure we have the state machine definition. Two states are shown in the figure. On the top left corner we find the initial state from where the state machine execution starts after the creation of an instance of *Comm_User*. From the initial state a transition labeled *initial* will take the state machine to the state *S1*. This initial transition does not require a

triggering event, so after creation of the *Comm_User* capsule instance the code on the transition is executed. It then waits in state *S1* for arrivals of messages. The code to be executed is shown in the right section of the same Figure 87. The code reads *TO_Device.releaseObject().send();* which is the format defined by ROSE-RT to send a signal from one capsule to another. The line is divided in three sections. The first one *TO_Device* indicates the name of the port to be used to send the signal. The middle section *releaseObject()* refers to the signal to be sent, while the last *send()* indicates the action to be performed.

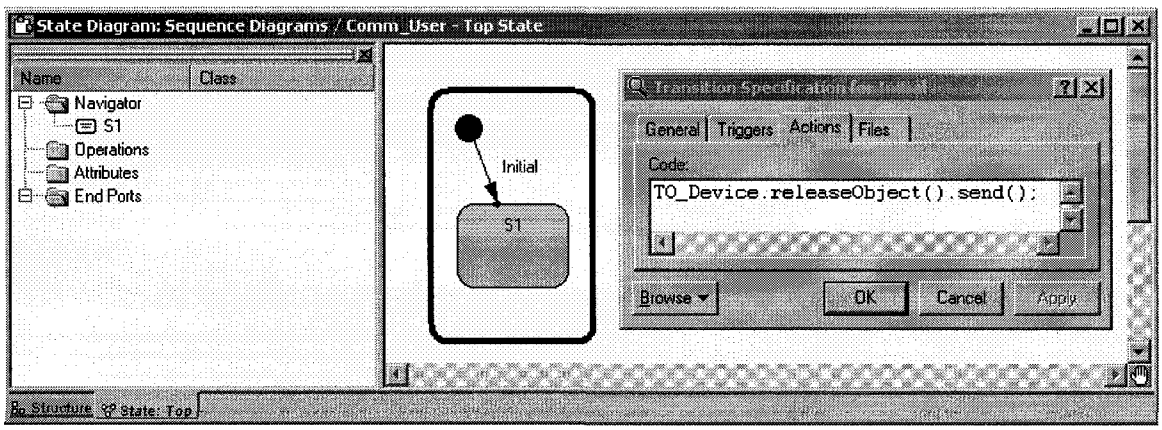


Figure 87. *Comm_User* state machine definition

The test fails and produces non-compliant points due to *required port not defined.* The reason being that the *TO_Device* port of the *Comm_User* structure is missing. The compiler detects and reports it. The compiler results are shown in Figure 88.

```
Comm_User.cpp
../src/Comm_User.cpp(51) : error C2065: 'TO_Device' : undeclared identifier
../src/Comm_User.cpp(51) : error C2228: left of '.releaseObject' must have class/struct/union type
    ../src/Comm_User.cpp(51) : error C2228: left of '.send' must have class/struct/union type
```

Figure 88. Compile errors: port not defined

Test 4: Design model missing a connector. This test is based on the new structural composition of the *Comm_User* capsule as shown in Figure 89. We can observe in the top center of the figure, that the *Comm_User* definition now defines a port *TO_Device.*

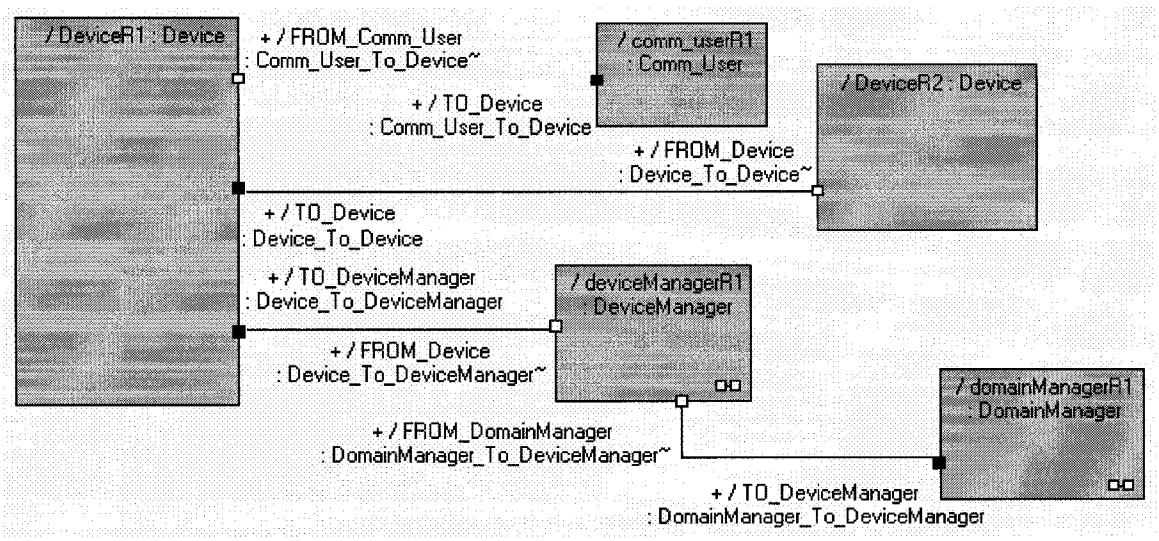


Figure 89. SWRadio Design structure diagram. TO_Device port added to Comm_User

The test fails and produces a non-compliant point due to *required connector not defined*. The reason being that there is no connector to act as a communication channel to pass a message from an instance of *Comm_User* towards an instance of *Device*. The compiler detects and reports it. The compiler results are similar to the ones shown in Figure 88 and thus we do not duplicate the figure.

Test 5: Design does not send three messages. This test is based on the structure defined in Figure 90. Notice here that the connector links the role *DeviceR1* of type *Device* with the Role *Comm_UserR1* of type *Comm_User*.

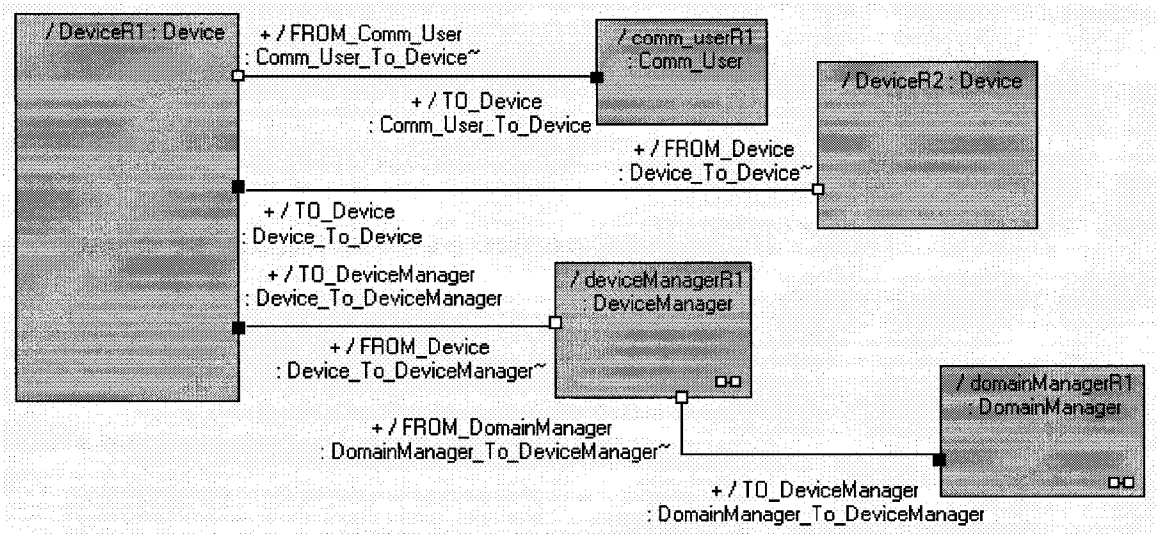


Figure 90. SWRadio Design structure diagram. Connector linking roles of Device and Comm_User

The test fails and produces non-compliant points due to *incorrect sending of messages*. Although the design execution successfully sent the *releaseObject* message (as shown in the middle of Figure 91) from the instance *Comm_UserR1* towards the instance *DeviceR1*, it failed to produce the same results for the three additional messages defined in the specification sequence diagram from Figure 80. RQA results are shown in Figure 92.

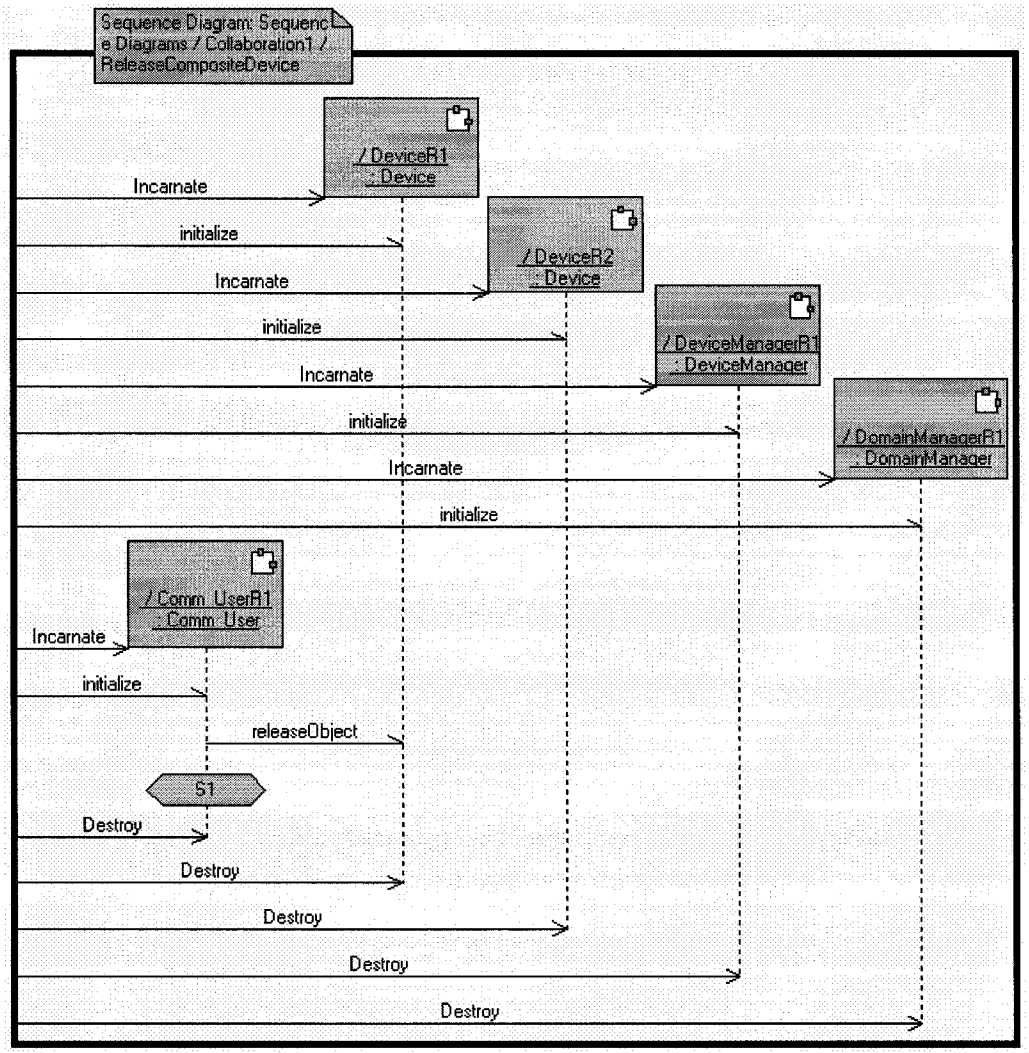


Figure 91. ReleaseCompositeDevice execution sequence diagram. releaseObject message sent

```
11:04:03| Differences between specification: "ReleaseCompositeDevice" and trace:
"ReleaseCompositeDevice_Trace_040216110400":
11:04:03| Warning:   Specification:   " : releaseObject" was unexpected
11:04:03| Warning:   Specification:   " : unregisterDevice" was unexpected
    11:04:03| Warning:   Specification:   " : unregisterDevice" was unexpected
```

Figure 92. ReleaseCompositeDevice test errors: 3 messages not sent/received

Test 6: Design does not send two messages. This test is based on the state machine for the *Device* capsule as shown in Figure 93. The state machine defines that while being in the state *S1* and upon reception of the *relaseObject* message through the port *FROM_Comm_User*, the transition from *S1* to *S2* will be triggered and code defined

inside the transition will be executed. The second transition defined in Figure 93 is triggered by the reception of a *releaseObject* message, but this time when the message arrives through the *FROM_Device* port.

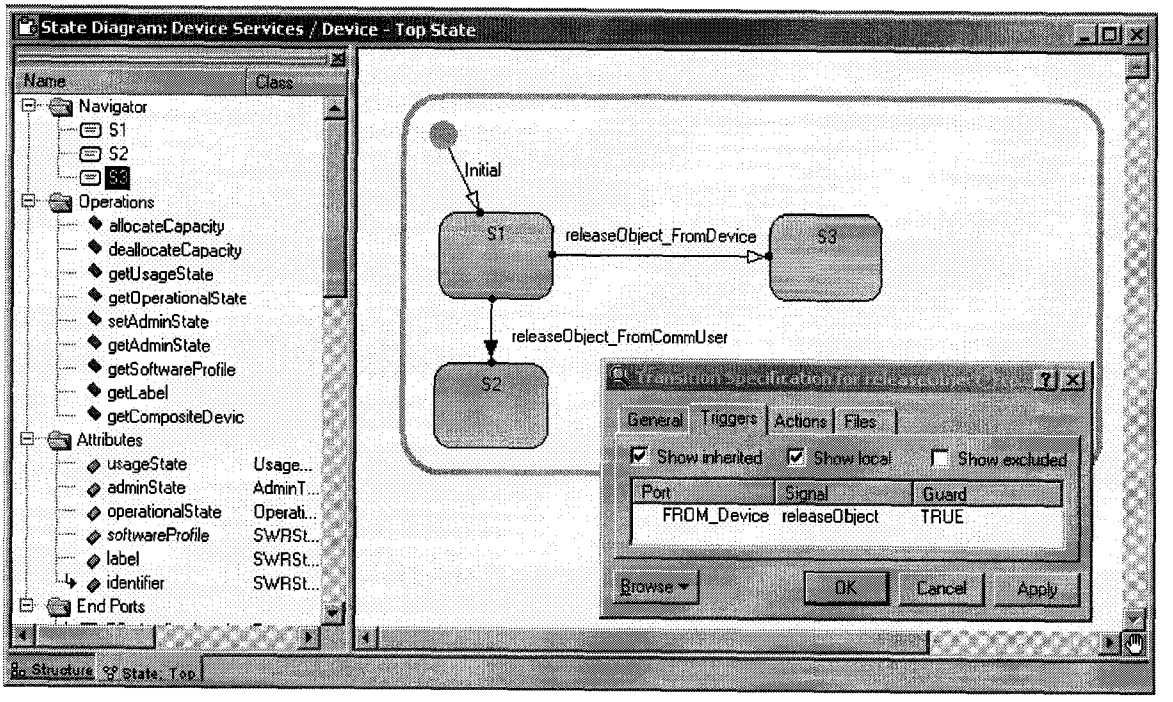


Figure 93. Device state machine definition. releaseObject message from different sources

The test fails and produces non-compliant points due to *incorrect sending of messages*. The results show that the model is successfully sending and receiving the first two messages and fails to send the last two (see Figure 94 and Figure 95).

```
11:44:36| Differences between specification: "ReleaseCompositeDevice" and trace:
"ReleaseCompositeDevice_Trace_040216114432":
11:44:37| Warning:  Specification:  " : unregisterDevice" was unexpected
11:44:37| Warning:  Specification:  " : unregisterDevice" was unexpected
```

Figure 94. ReleaseCompositeDevice test errors: 2 messages not sent/received

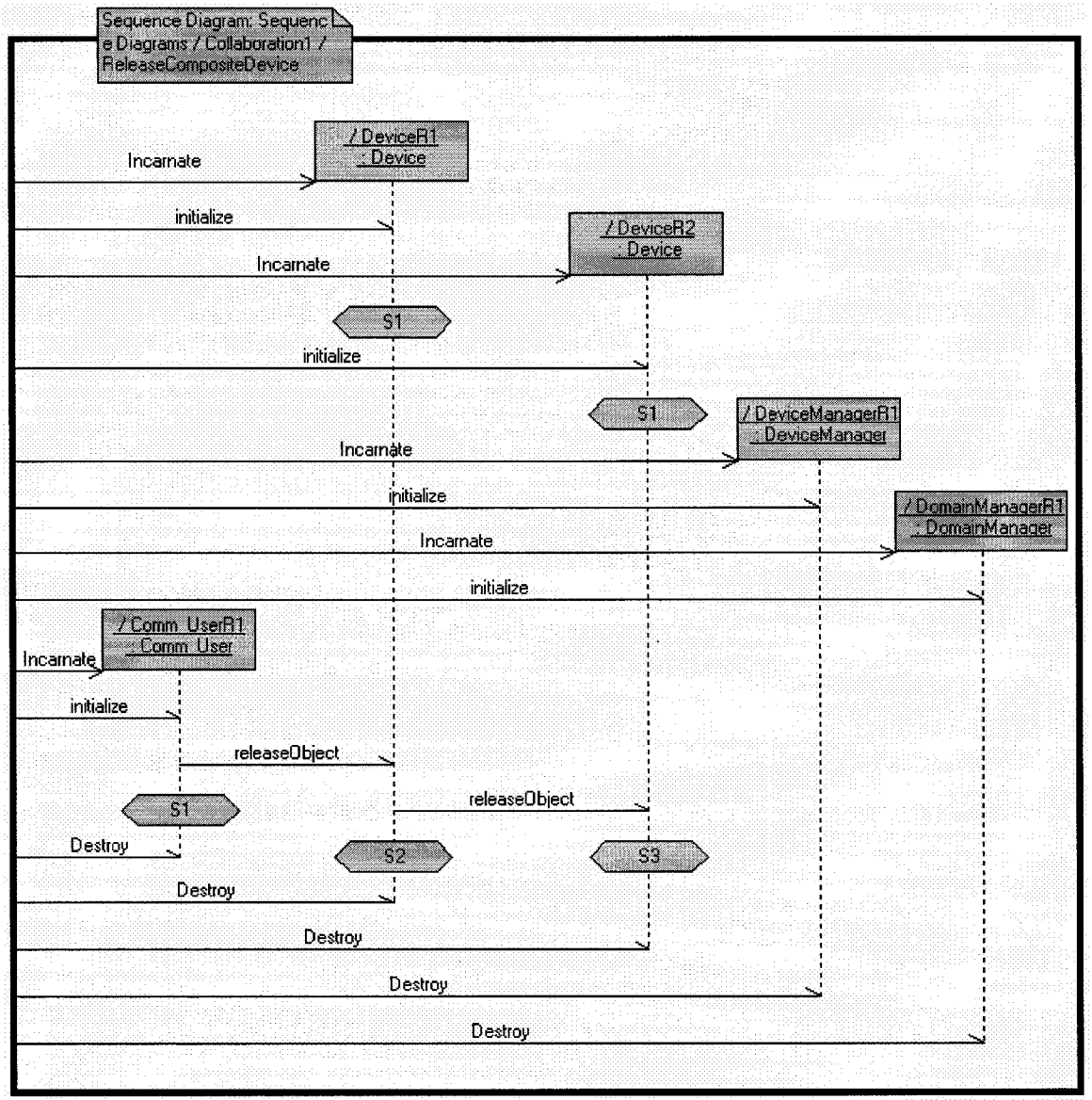


Figure 95. ReleaseCompositeDevice execution sequence diagram. Two releaseObject messages sent

Test 7: Design missing data to be attached to a message. This test is based on the code shown in Figure 96, in which the signal *unregisterDevice* is to be sent using the *TO_DeviceManager_As_creator* port. It also considers the definition of the protocol class shown in Figure 97.

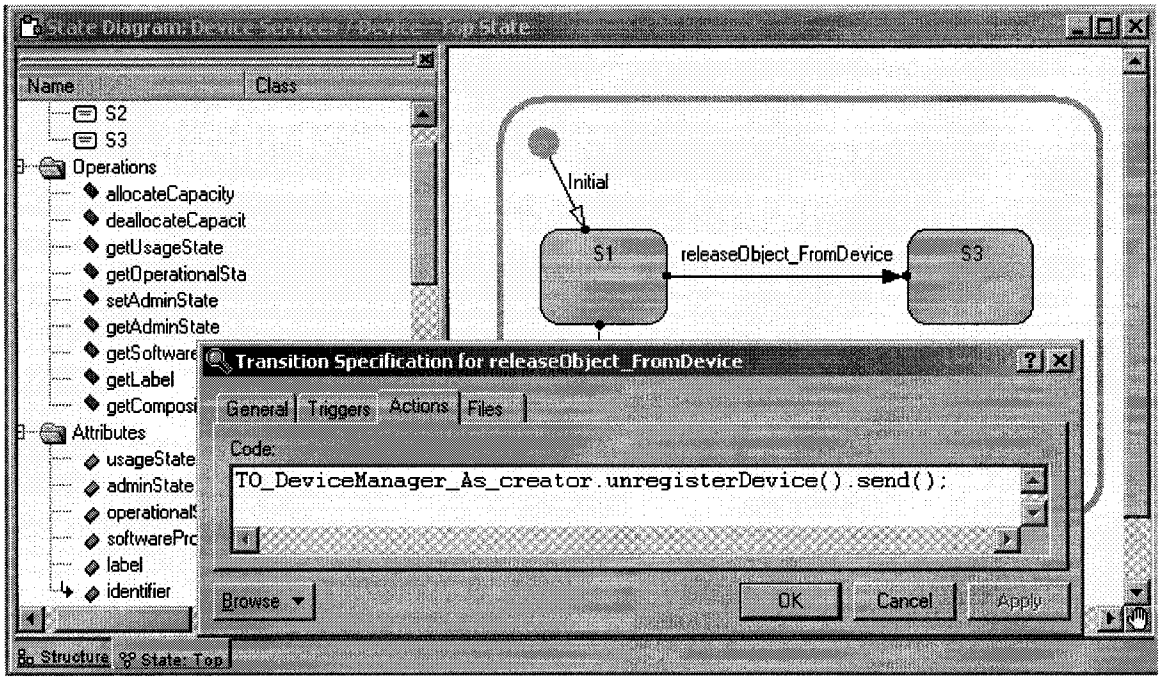


Figure 96. Device state machine definition. unregisterDevice message to be sent

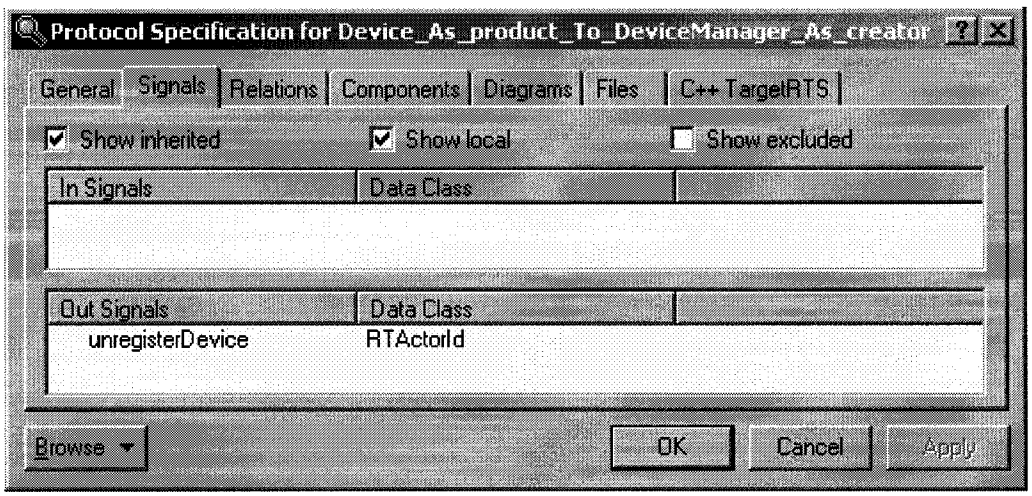


Figure 97. Device_As_product_To_DeviceManager_As_creator protocol definition

The test fails and produces non-compliant points due to *missing or incorrect data attached to a message*. The error messages are shown in Figure 98 and are related to the protocol definition presented in Figure 97. The signal *unregisterDevice* is defined to carry data attached to it as defined in the *Device_As_product_To_DeviceManager* from Figure 97.

```
!> Compiling Device
Device.cpp
../src/Device.cpp(702) : error C2660: 'unregisterDevice' : function does not take 0 parameters
../src/Device.cpp(702) : error C2228: left of '.send' must have class/struct/union type
        NMAKE : warning U4010: 'Device.OBJ' : build failed; /K specified, continuing ...
```

Figure 98. Compile errors: data missing in signal

Test 8: Design with incompatible data to be attached to a message. This test is based on the state machine defined in Figure 99. The design attempts to send an *int* value attached to the *unregisterDevice* signal through the *TO_DeviceManager_As_creator* port.

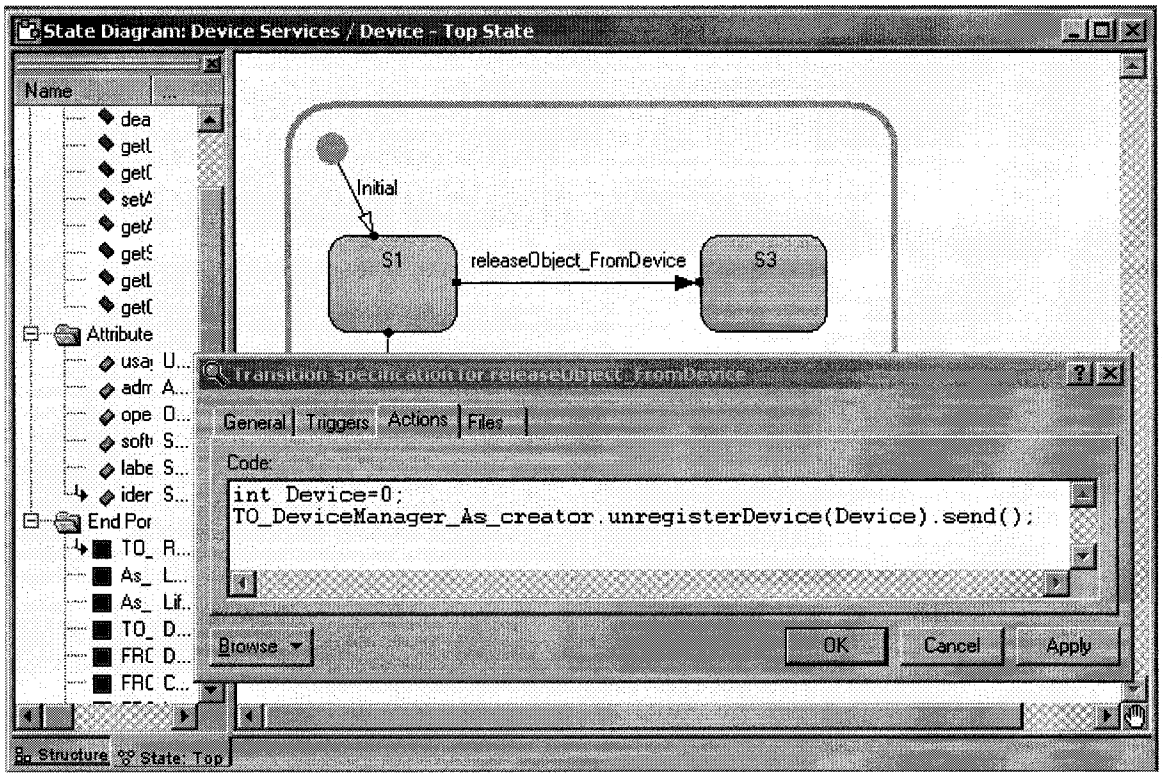


Figure 99. Device state machine definition. unregisterDevice message to be sent with data attached to the signal

The test fails and produces non-compliant points due to *missing or incorrect data attached to a message*. Test results are shown in Figure 100 in which incompatible types are used between the one attempted to send (an *int* type) and the one defined in the protocol class (a *RTActorId* type) from Figure 97.

```
Device.cpp
../src/Device.cpp(703) : error C2664: 'unregisterDevice' : cannot convert parameter 1 from 'int' to 'const struct
RTTypedValue_RTActorId &'
        Reason: cannot convert from 'int' to 'const struct RTTypedValue_RTActorId'
        No constructor could take the source type, or constructor overload resolution was ambiguous
../src/Device.cpp(703) : error C2228: left of '.send' must have class/struct/union type
NMAKE : warning U4010: 'Device.OBJ' : build failed; /K specified, continuing ...
```

Figure 100. Compile errors: incorrect data type of data attached to the signal

Test 9: Design does not send one message. This test is based on the state machine shown in Figure 101 in which a *RTActorId* type of value is attached to the signal to be sent.

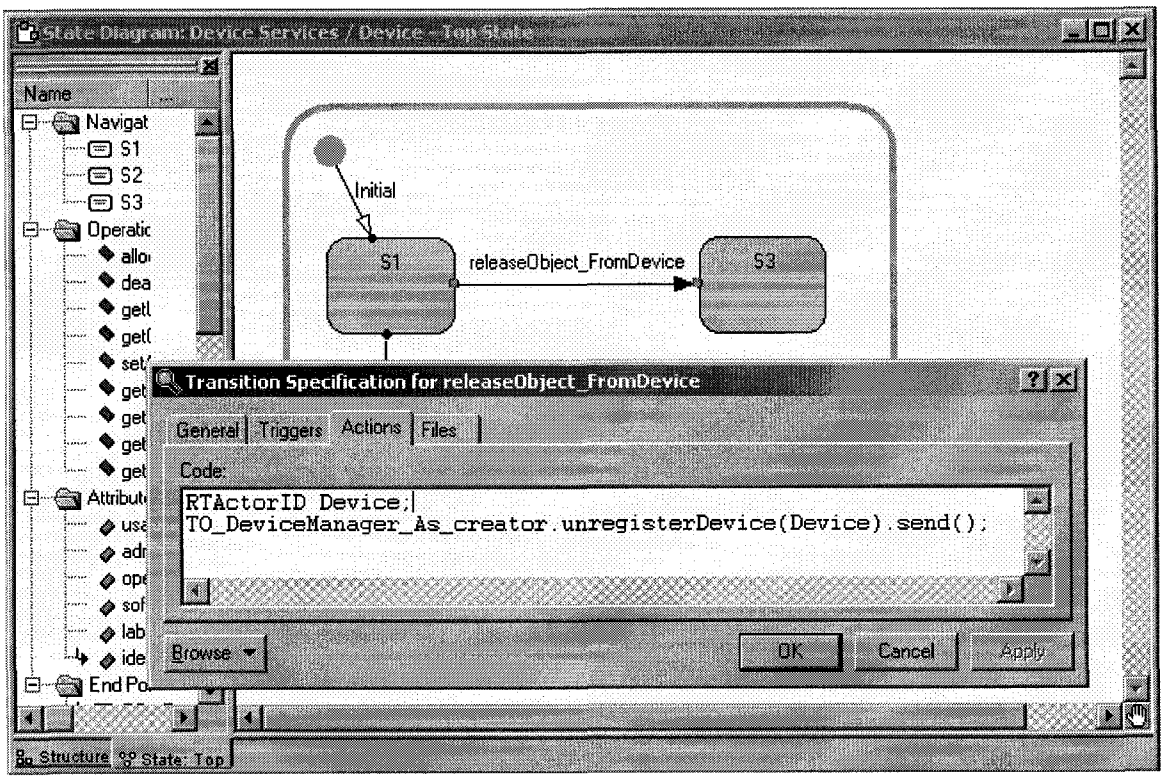


Figure 101. Device state machine definition. unregisterDevice message to be sent with data attached to the signal with type as defined in the protocol specification

The test fails and produces non-compliant points due to *incorrect sending of messages.* Although the test execution produces three correct sent messages, one message is not sent (see Figure 102).

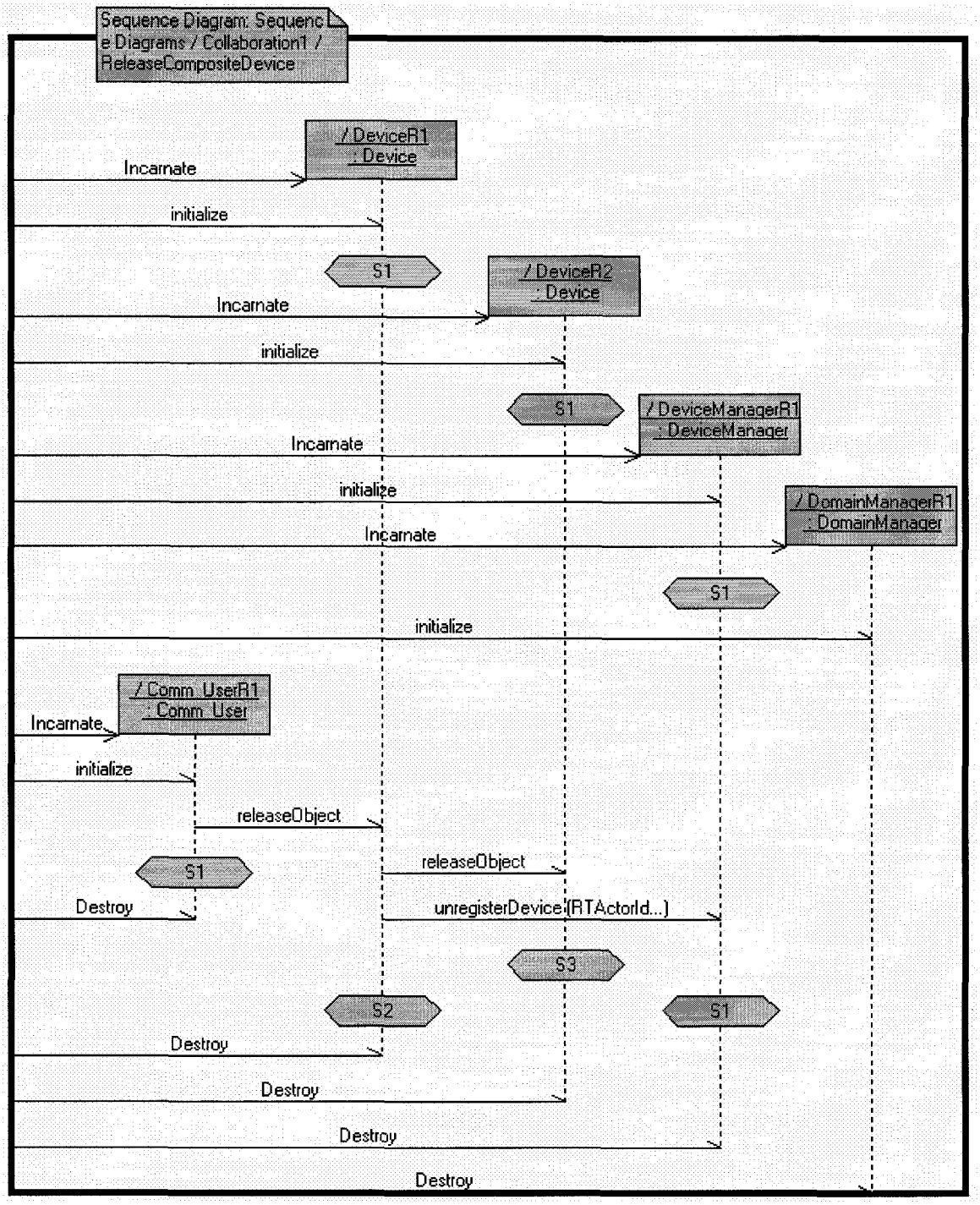


Figure 102. ReleaseCompositeDevice execution sequence diagram. Two releaseObject and one unregisterDevice messages sent

Test 10: Ready to execute ReleaseCompositeDevice scenario. This test is based on Figure 103 in which the data received by the signal (expressed as **rtdata*) is relayed towards the *DomainManager* through the *FROM_DomainManager_As_registrar* port.

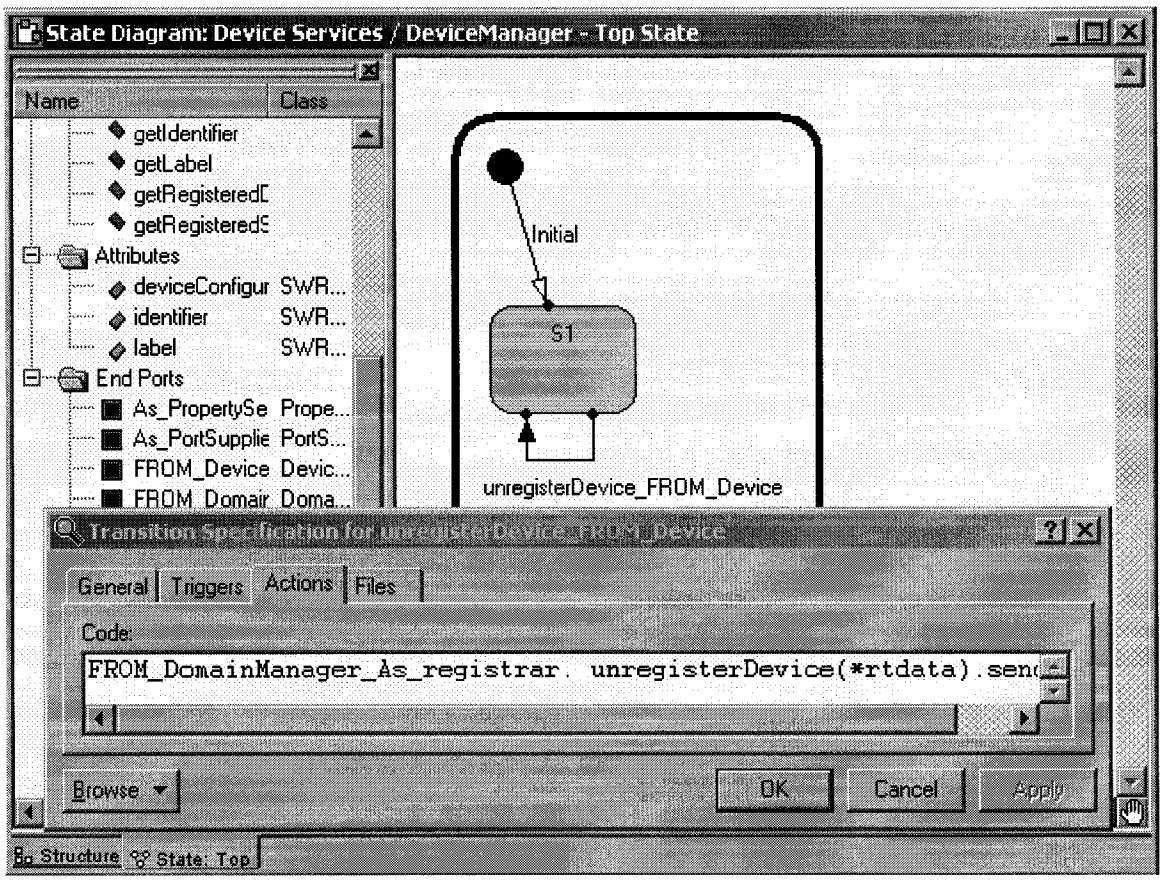


Figure 103. DeviceManager state machine definition.

The test results in a successful execution as shown in Figure 104: *Number of test that passed: 1.*

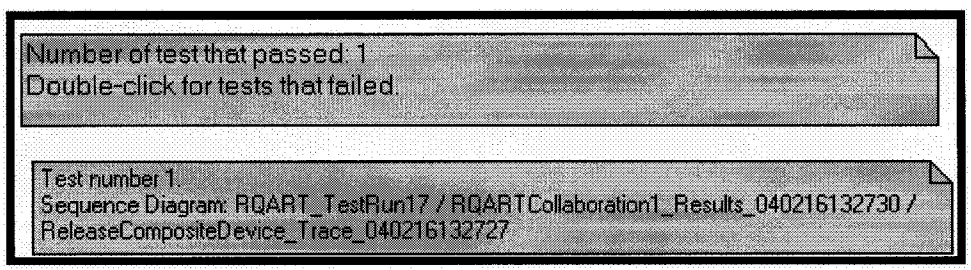


Figure 104. ReleaseCompositeDevice test results: passed

Figure 105 shows the automatically generated sequence diagram after execution. It shows the messages being sent/received as defined in the specification sequence diagram of Figure 80.

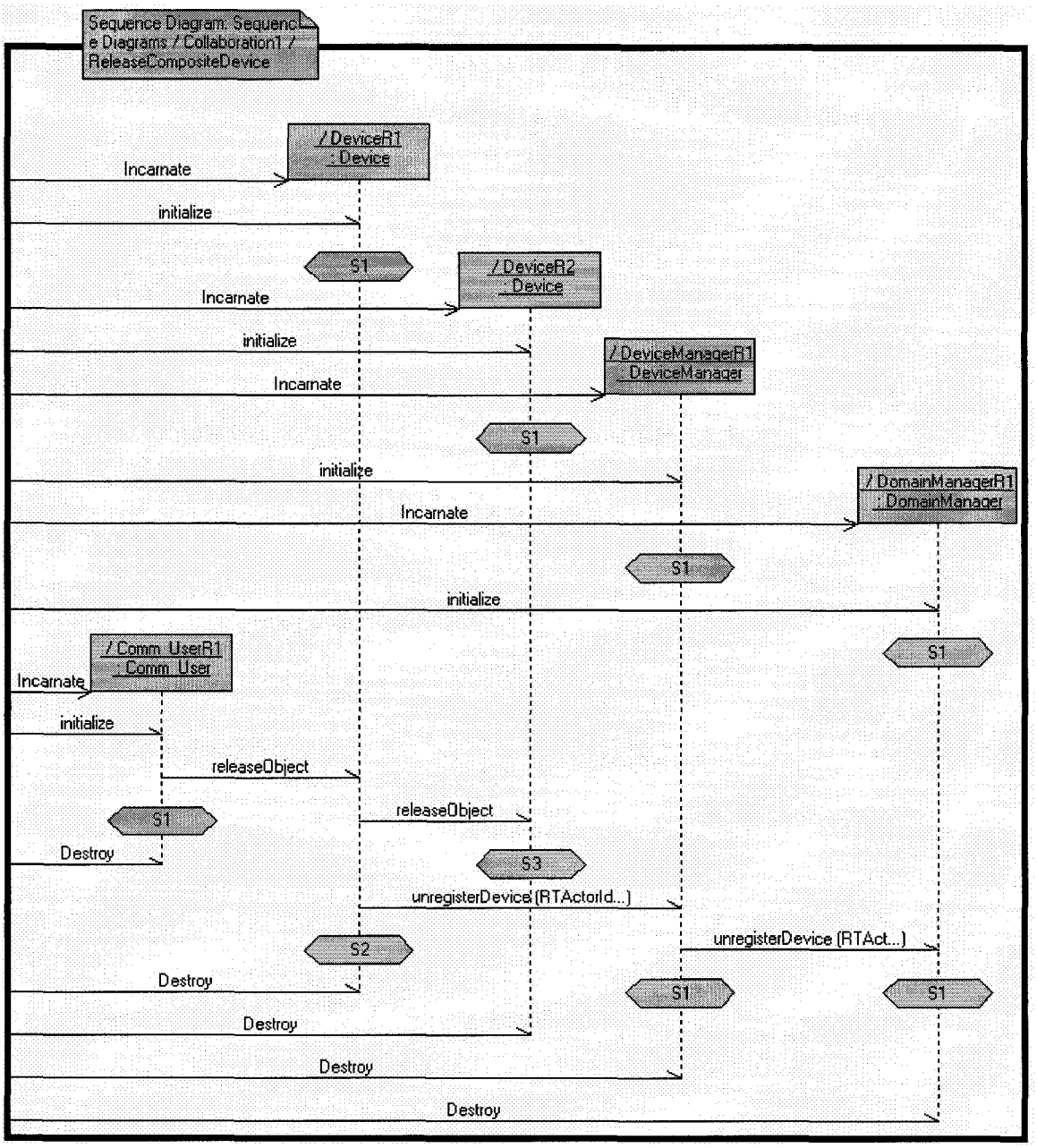


Figure 105. ReleaseCompositeDevice correct execution sequence diagram

ReleaseCompositeDevice sequence diagram compliance verification.

The next set of test cases relates to the verification of a second sequence diagram from the SWRadio specification: the *releaseAggregatedDevice* sequence diagram shown in Figure 81.

The *releaseCompositeDevice* and *releaseAggregatedDevice* sequence diagrams both use the same instance definitions but differ in the usage of messages being sent/received. In the *releaseAggregatedDevice* sequence diagram from Figure 81, it is the *aggregated* instance (being played by the role *DeviceR2*) of the *Device* capsule that receives the *releaseObject* message from the *comm_user* instance. In the *releaseCompositeDevice* it is the *composite* instance being played by role *DeviceR1* of the *Device* capsule that receives the same signal. Also in the *releaseAggregatedDevice* sequence diagram, the *removeDevice* signal that is sent from the *aggregated* instance to the *composite* instance is not used in the *releaseCompositeDevice* sequence diagram from Figure 80.

Test 11: Not ready for releaseAggregatedDevice scenario. This test is based on the same design model use for the previous test. The test fails and produces non-compliant points due to *incorrect sending of messages*. Results are shown in Figure 106. The execution sequence is shown in Figure 108. The reason for the failure is that some messages from the specification are not handled correctly, while some others from the execution appear there without being part of the specification. The differences are shown in Figure 107.

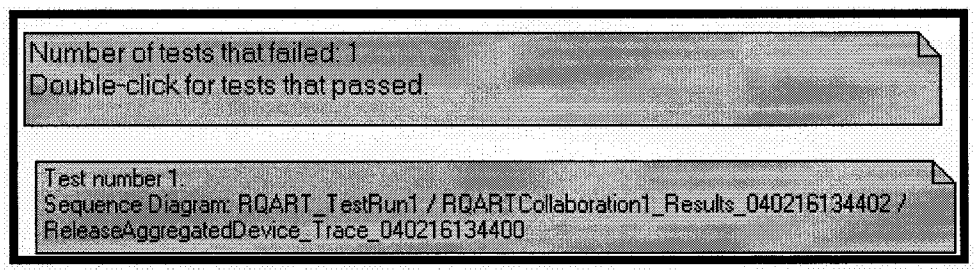


Figure 106. ReleaseAggregatedDevice test results: failed. No behavior defined to handle second scenario

```
13:44:03| Differences between specification: "ReleaseAggregatedDevice" and trace:
"ReleaseAggregatedDevice_Trace_040216134400":
13:44:03| Warning:   Specification:   " : releaseObject" was unexpected
13:44:03| Warning:   Trace:   " : releaseObject" was unexpected
13:44:03| Warning:   Specification:   " : removeDevice" was unexpected
13:44:03| Warning:   Trace:   " : releaseObject" was unexpected
13:44:03| Warning:   Trace:   " : unregisterDevice" was unexpected
13:44:03| Warning:   Specification:   " : unregisterDevice" was unexpected
```

Figure 107. ReleaseAggregatedDevice test errors: unexpected messages in both specification and execution sequence diagrams

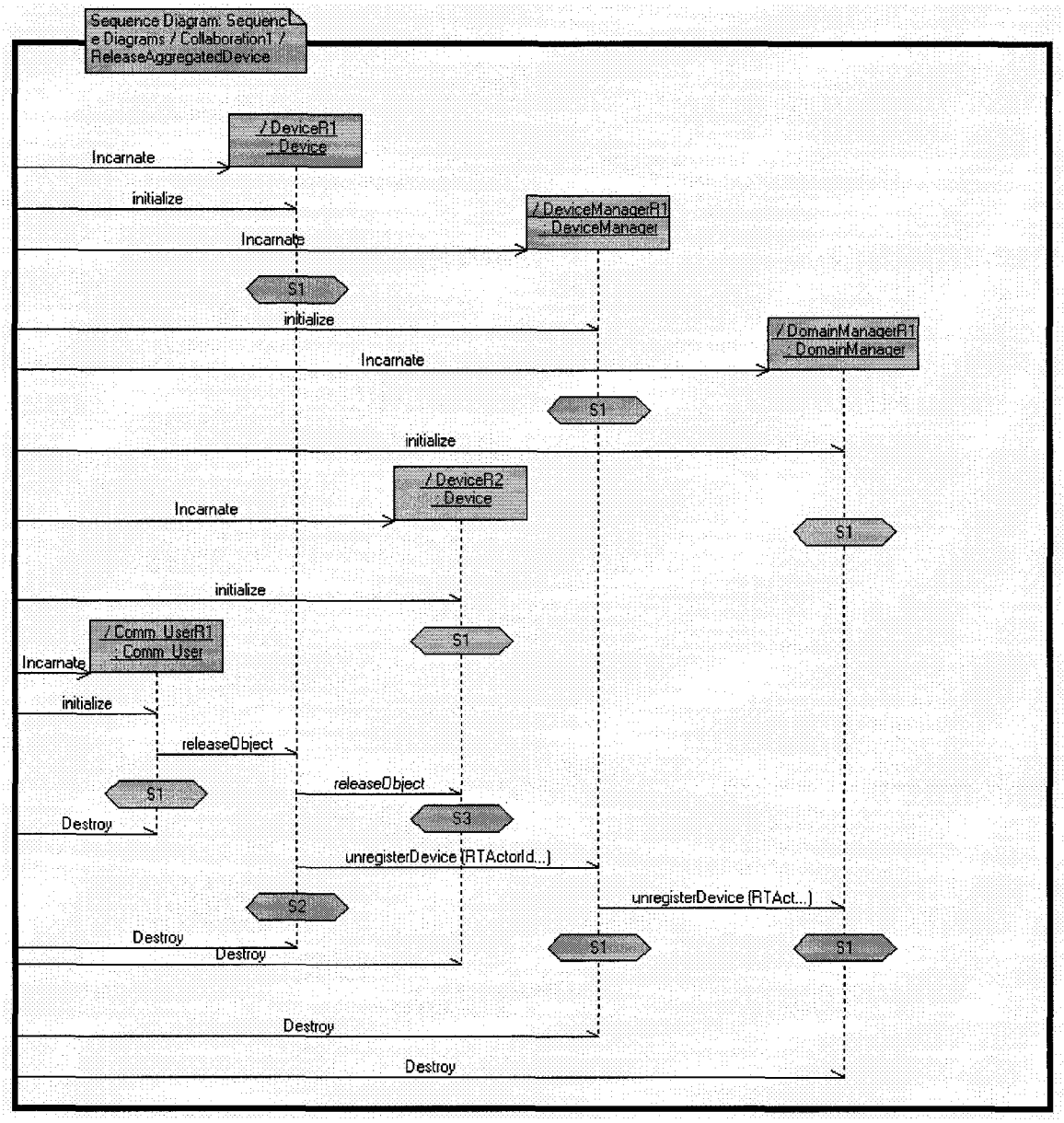


Figure 108. ReleaseAggregatedDevice execution sequence diagram. Behavior shown as for releaseCompositeDevice scenario

Test 12. Ready to execute releaseAggregatedDevice scenario. This test is based on the state machine of the *Device* capsule as shown in Figure 109. The *Device* capsule is capable of handling both the *releaseCompositeDevice* and the *releaseAggregatedDevice* sequence diagrams. We show the successful test results for the *releaseAggregatedDevice* scenario in Figure 110.

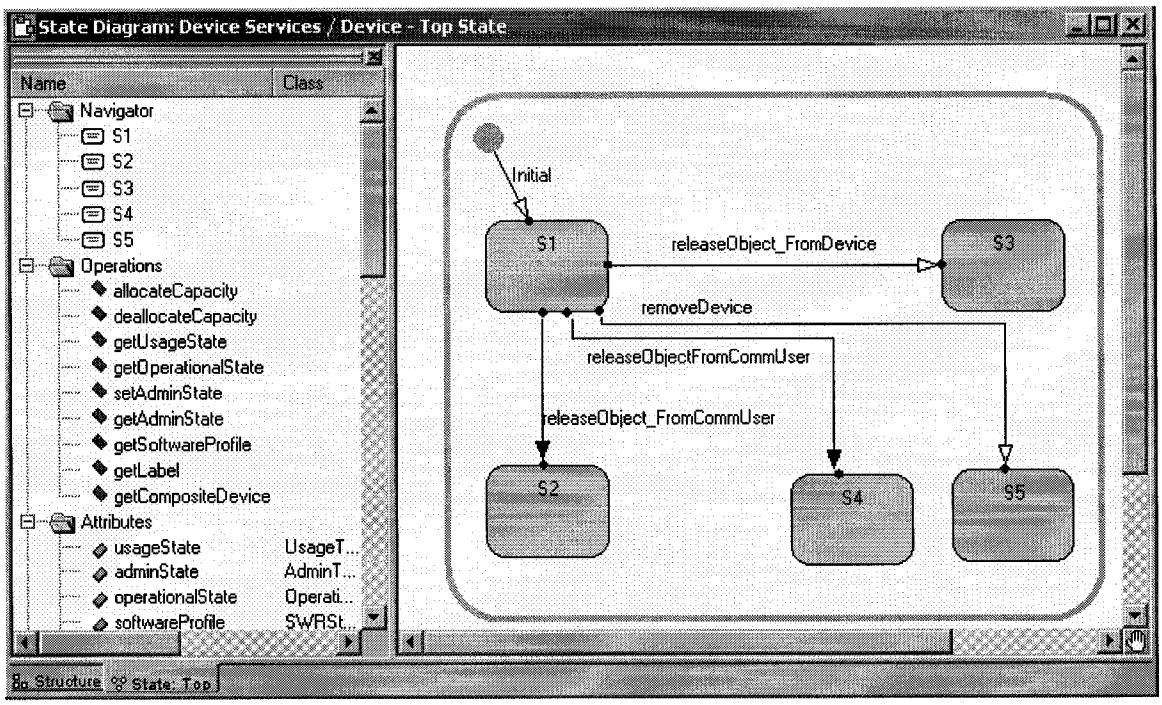


Figure 109. Device state machine definition. Ready to handle both scenarios (independent from each other)

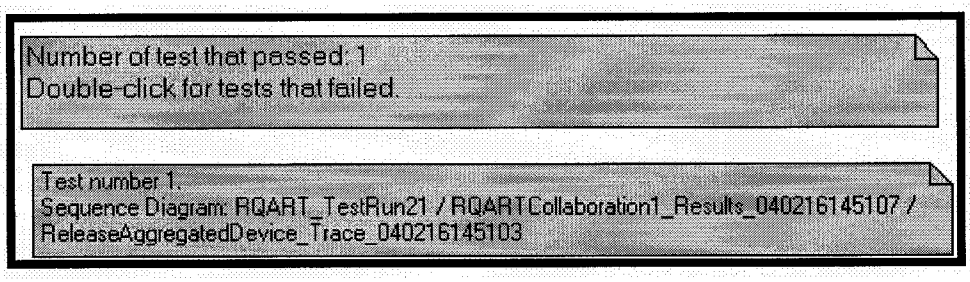


Figure 110. ReleaseAggregatedDevice test results: passed

Figure 111 presents the actual execution sequence generated by RQA for the releaseAggregatedDevice scenario.

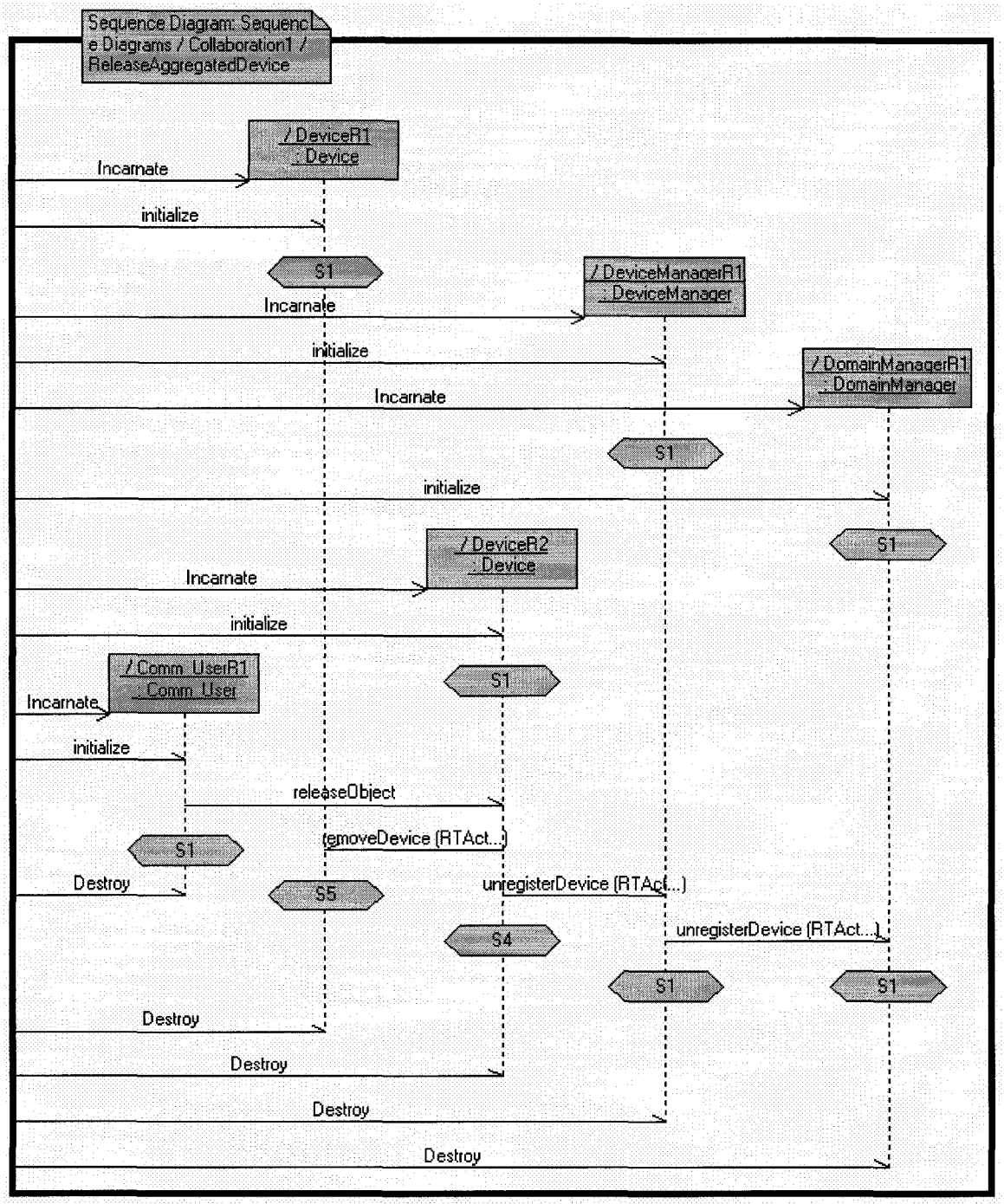


Figure 111. ReleaseAggregatedDevice correct execution sequence diagram

ReleaseCompositeDevice and releaseAggregatedDevice linked scenario compliance verification.

Tests 13. Not ready to execute releaseAggregatedDevice scenario immediately after ReleaseCompositeDevice scenario. This test uses the same design as in test 12,

but verifies for compliance the execution of two consecutive scenarios. The test attempts to execute the *releaseAggregatedDevice* immediately after *releaseCompositeDevice*. The test result of such an attempt is that the first will succeed but the second will not (see Figure 112). This is a non-compliant point due to *multi scenario execution* (see section 3.2.2).

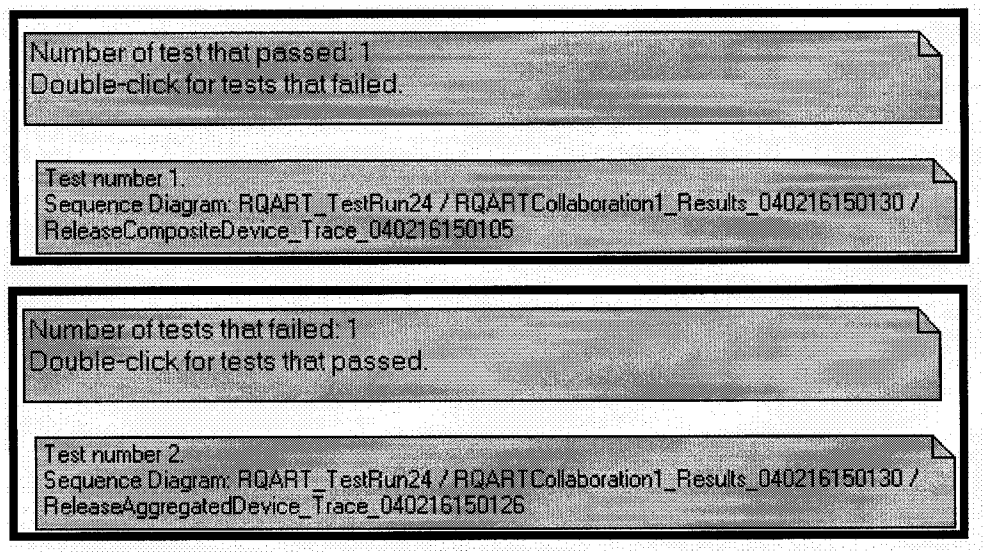


Figure 112. ReleaseCompositeDevice and releaseAggregatedDevice linked test results: 1 pass 1 failed

Test 14. Ready to execute releaseAggregatedDevice scenario immediately after ReleaseCompositeDevice scenario. This test is based on the state machine shown in Figure 113.

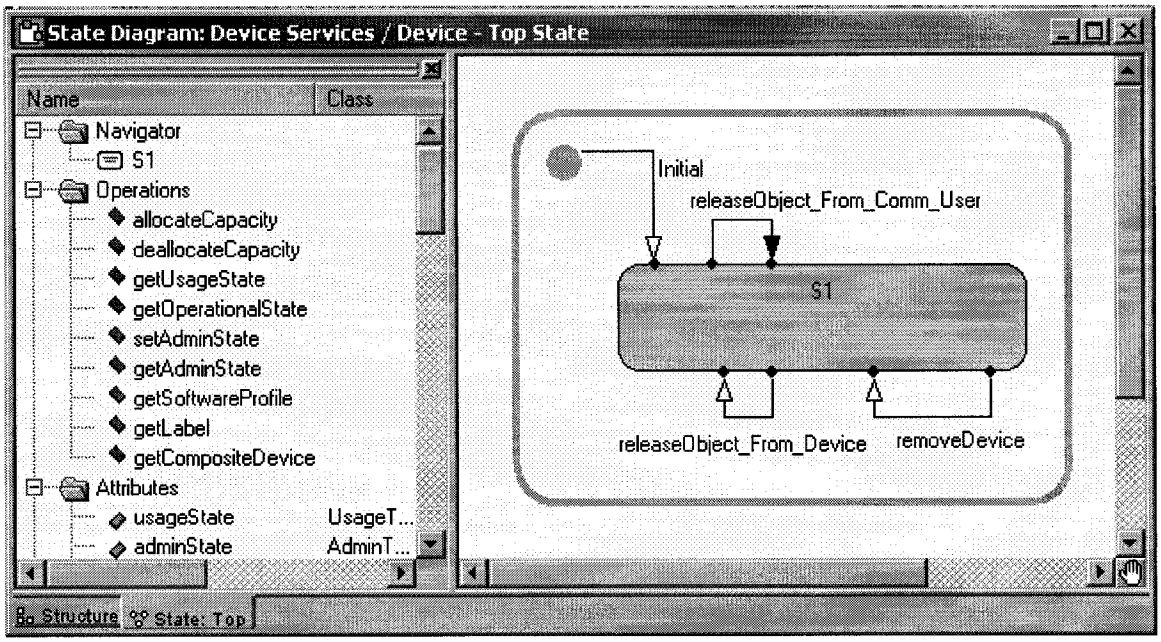


Figure 113. Device state machine definition. Ready to handle both scenarios (with execution one after the other)

The test results in a successful execution as shown in Figure 114.

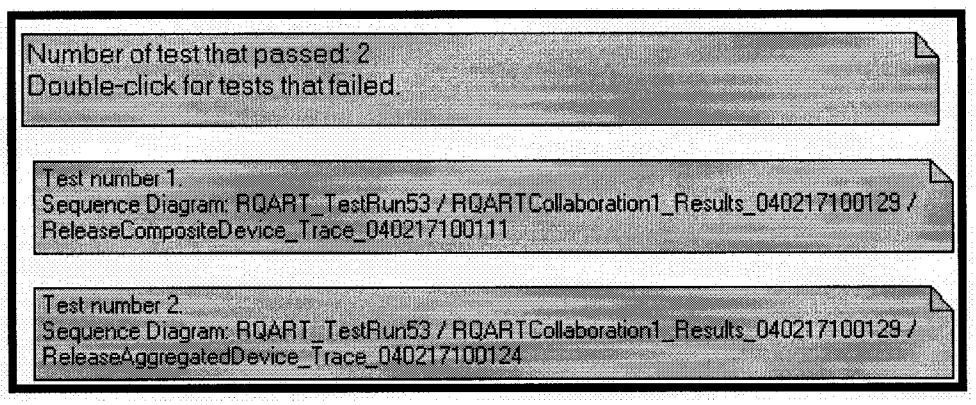


Figure 114. ReleaseCompositeDevice and releaseAggregatedDevice linked test results: 2 test passed

Test 14 concludes the behavioral compliance section of the SWRadio case study. The next section addresses future challenges to handle a newer version of the SWRadio PIM.

4.2 Challenges towards a new Software Radio PIM

The SWRadio specification is going through a major review process in which we expect a SWRadio standard specification to be produced. In this section, we present a high level overview of the tasks and challenges we identify in order to adapt the work that has already been done, and apply our approach to the new SWRadio PIM specification. However, we do not define new mappings nor consider the application of the two-step approach for compliance verification to this new specification. Such an endeavor is considered part of the future work related to this research.

This section has been divided to address different topics related with the new SWRadio specification. Section 4.2.1 presents the contents of the new SWRadio specification, which is defined in two separate models: The first defines a SWRadio PIM, while the second does so for a SWRadio PSM. Section 4.2.2 analyses the challenges that the new SWRadio Profile imposes on our current work if we were to reuse part of the work already done. Finally, section 4.2.3 presents potential new mappings to define a complete set of mappings for the new SWRadio Specification.

4.2.1 SWRadio *Spec* contents

In this section we present a high level overview of the contents of the new SWRadio Specification. We first describe its top-level package structure, to later present more low-level details about the elements and relationships that compose the models.

Figure 115 illustrates the top-level packages of the new SWRadio specification. On the left-hand side of the figure, the top-level packages of the SWRadio PIM *Spec* are defined. The right-hand side of Figure 115 does so for the SWRadio PSM *Spec*. The figure describes dependency relationships between the top-level packages of both models.

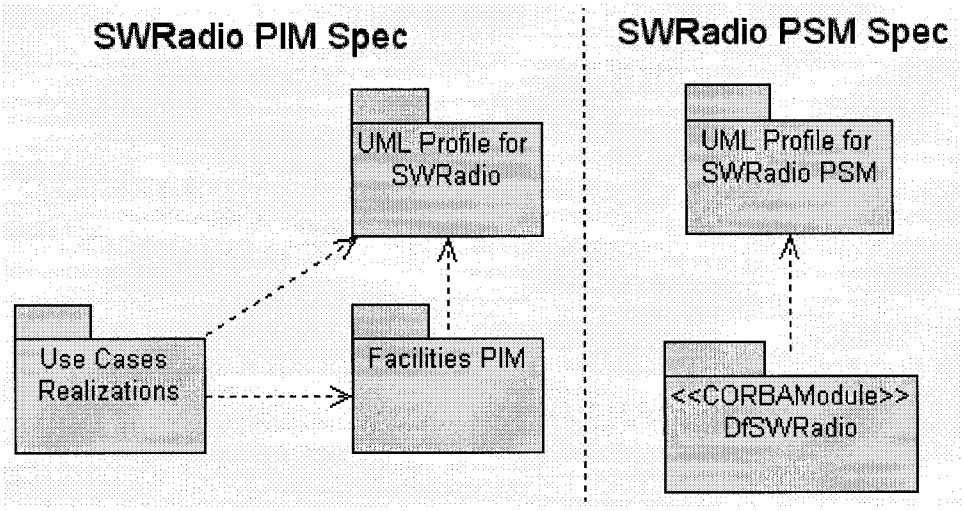


Figure 115. New SWRadio *Spec* Top-Level Packages

For the SWRadio PIM *Spec*, Figure 115 describes how the Facilities PIM and Use Case realization packages depend on the constructs defined in the UML Profile for SWRadio package. The Facilities PIM package defines a SWRadio model that is independent of an execution platform. The Use Case Realizations package provides behavioral definition for elements in the Facilities PIM package by means of: 1) use case diagrams that express use case realizations, and 2) sequence diagrams that specify the details of each realization. For the SWRadio PSM *Spec*, Figure 115 describes how the DfSWRadio, a CORBA specific PSM, depends on the constructs defined in the UML Profile for SWRadio PSM package.

As our previous work was done with a platform independent model, we will continue our discussion considering only the SWRadio PIM *Spec* on the left-hand side of Figure 115. Figure 116 presents the top level-packages that compose the SWRadio Facilities PIM package. The new top-level packages shown in Figure 116 differ considerably from the ones described in section 2.3.3 (Figure 19) for the first version of the SWRadio PIM on which we based our work. The difference does not pose a problem though. Recall from 2.2.5, that model transformation is performed at MOF metalevel M2, which is concerned with the entities/relationships (metamodel) used to define software models like the new SWRadio PIM. Thus variation of the content of the SWRadio PIM does not pose by itself a challenge to our approach. On the other hand, the types of elements and

relationships used in the new SWRadio PIM directly impact on the nature of our mappings. Finally, the use given to such elements/relationships of the SWRadio PIM are also considerably important to our work. We further discuss these issues in section 4.2.3 below.

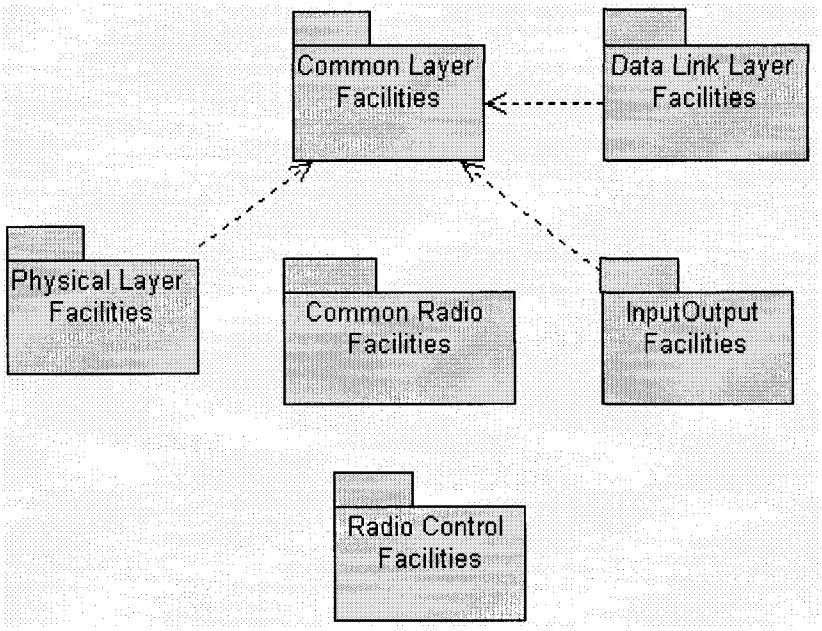


Figure 116. Facilities PIM Top-Level Packages

With respect of the composition of the new SWRadio PIM (in terms of types and numbers of elements and relationships) Table 45 presents a comparison between the first version of the SWRadio PIM and the new version of the specification. The first column presents the name of the element/relationship. The second column presents the number of element/relationships of each type present in the first version of the SWRadio PIM. The third column does so for the new (Final Adopted) version of the SWRadio PIM. Finally, the fourth column presents the percentage growth when comparing the two models.

Table 45. Comparison between the first and the final adopted versions of the SWRadio PIM

Element/Relationship	SWRadio PIM First version	SWRadio PIM Final Adopted	Variation
packages	43	191	344 %
classes	221	672	204 %
interfaces	19	55	189 %
operation	112	273	144 %
parameters	84	258	207 %
attributes	271	882	225 %
simple associations	22	349	1486 %
Aggregations	20	18	-10 %
compositions	102	30	-71 %
generalization	60	197	228 %
realize	18	106	489 %

From Table 45 we can observe that the new version has considerably grown from the first one although, we repeat, the number of times elements/relationships are used within a specification does not represent a challenge to our work. We have even applied our transformation tool to the new specification and we have *partially succeeded* in generating a *tSpec* model. We emphasized *'partially succeeded'* in the last sentence, because although a *tSpec* model was generated into a target ROSE-RT metamodel, not all the elements/relationships from the new SWRadio PIM were used as originators of elements/relationships of the *tSpec* model. This is because only mappings defined and implemented from the first version of the SWRadio PIM were applied by our transformation tool. There is the need to define more mappings to successfully generate a semantically equivalent *tSpec* model for the new SWRadio PIM specification. We discuss about those new mappings in section 4.2.3.

The next section presents the challenges we face when considering a SWRadio Profile being used by the new SWRadio PIM specification

4.2.2 SWRadio Profile

In this section we discuss the challenges that a SWRadio Profile being used by the new SWRadio PIM specification brings to our work. We first start by making a high level comparison between the two versions of the SWRadio PIM (with respect of the use of the new profile), and subsequently propose possible solutions.

Recall from 2.3.3 that the first version of the SWRadio PIM was built using standard UML defined elements and relationships. In other words, there was no SWRadio-specific element or relationships used in defining the model. For the new version though, on top of the same set of UML elements and relationships used in the first version, there is a new set of SWRadio-specific elements like *EventChannels*, *commchannels*, *services*, *devicecomponents*, *radiomanagers* and many more that define their own semantics. The good news is that those new SWRadio-specific types of elements are also defined in terms of basic UML elements. Still, there is an implicit relationship that is hidden from the physical observation of the models. Let us take, for example, the *AntennaDevice* element shown in the bottom left area of Figure 117. The figure shows the relationships that the *AntennaDevice* element possesses with other elements of the specification. The figure defines that the *AntennaDevice* specializes the *RFIFComponent*, and realizes the *IPolarization* and *IRadiationPattern* elements. But the Figure 117 also shows that the AntennaDevice element is of a stereotype *devicecomponent*. In other words, the *AntennaDevice* besides being a UML Class, is also a SWRadio *DeviceComponent*[12].

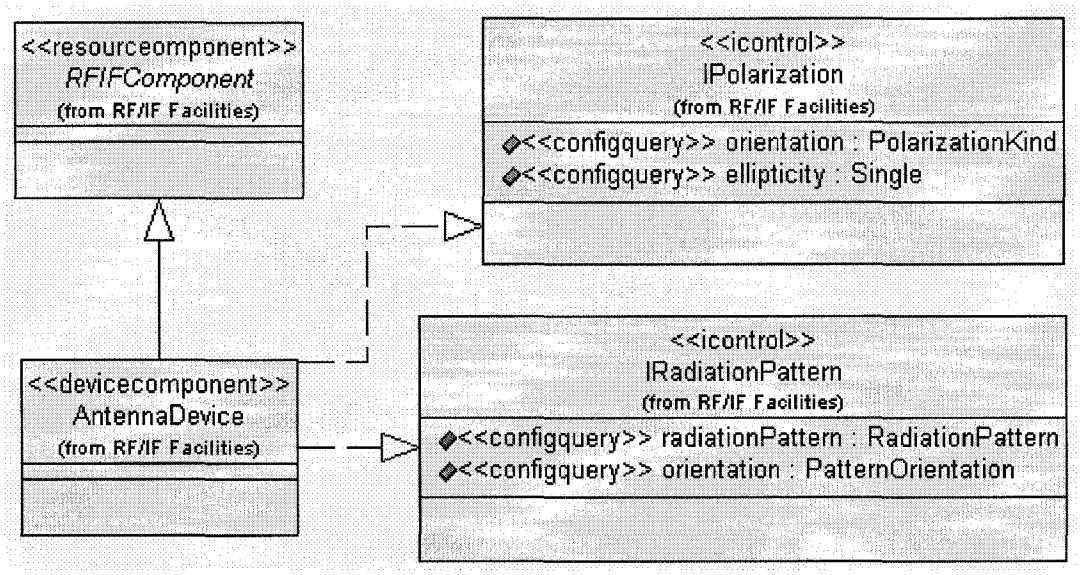


Figure 117. SWRadio AntennaDevice definition (stereotype example)

[12] The difference between the words 'devicecomponent' (all lowercase) and 'DeviceComponent' (with capitals 'D' and 'C') responds to a modeling convention by the Software-Based Communication Domain Task Force. In both cases they refer to the same element.

From Figure 117 we can conclude then that besides the properties and relationships shown in Figure 117, the *AntennaDevice* also possesses the properties and relationships defined by the *DeviceComponent* stereotype in Figure 118[13]. The *DeviceComponent* is the one at the bottom (center-left) of the figure, and there is no need to try to understand the relationships it has with the other elements in the figure. Suffice it to observe that by being a type of *DeviceComponent*, the *AntennaDevice* from Figure 117 suddenly becomes a little bit more complex that what appeared at first glance.

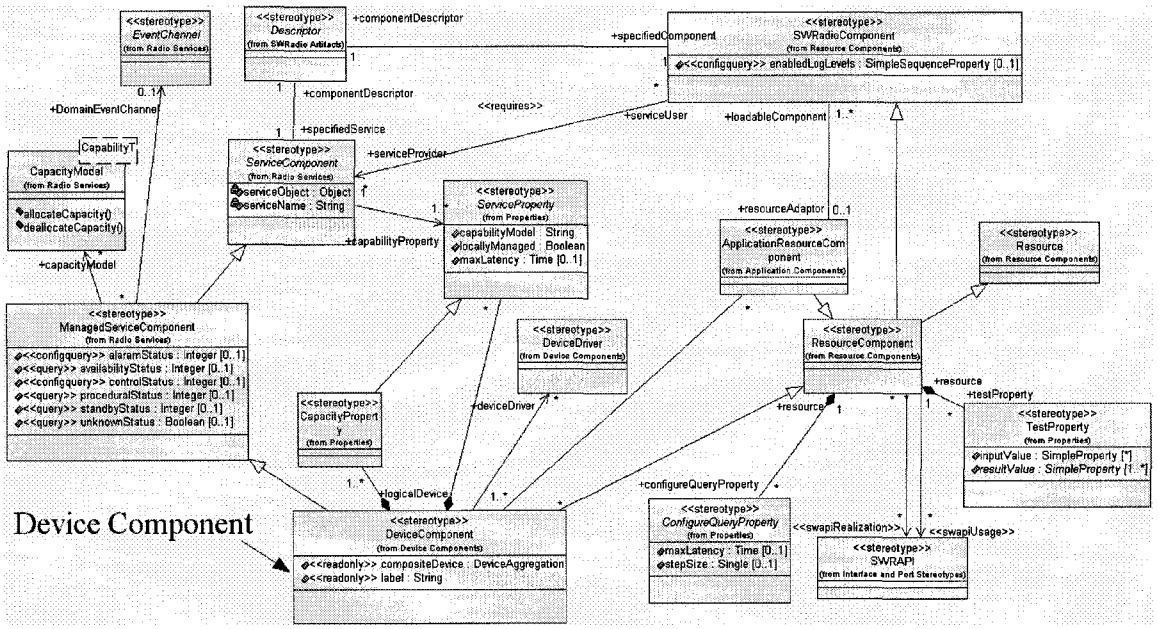


Figure 118. SWRadio DeviceComponent definition (stereotype example)

Besides the semantics attached to the *DeviceComponent*, we must recognize the fact that we have not defined a mapping to deal with specification-specific stereotypes. There are two possible quick fixes to our problem, both in the form of new mappings. In the first solution a new *StereotypeMapping* copies the stereotype of elements in the *Spec* model into stereotypes of elements in the *tSpec* model. The second considers also a new *StereotypeMapping* that would treat stereotypes in the *Spec* model as generalization relationships. In other words, it places a generalization relationship between elements in the *Spec* and the elements they are a stereotype of (in our example AntennaDevice

[13] We apologize for the font size and clutter of Figure 118, but we are trying to make a point here.

specializing DeviceComponent). These two quick solutions would allow us to generate a *tSpec* model that would carry on with the semantics of stereotyping. None of them represent a lot of extra work... but there is a major issue that we have not raised yet: the actual definition of SWRadio Profile elements in the *Design* models we want to eventually verify for compliance. We discuss a possible solution in the following paragraphs.

Let us start by saying that most of the work we are about to discuss is still not mature and that the following discussion should perhaps be better addressed in the future work section at the end of this document. As it is strongly related to the topic at hand, we opted to promptly discuss it here. The problem we face now is not a matter of feasibility to carry over the semantics of stereotyping from the *Spec* model to the *tSpec* model. What we have to consider now is if elements from the SWRadio Profile are even being defined in the Design model. It is possible that a SWRadio PIM Design presented for compliance verification includes only PIM elements and not profile elements. PIM elements may contain all the required properties established by the SWRadio *Spec* (including those added by being a stereotype of an element defined in the SWRadio profile), but again, those profile elements may not be present in the model.

We abstract away from most of the relationships shown in Figure 118 (that relate to the DeviceComponent) and work with only a few of them. We do the same with all the relationships of the *AntennaDevice* definition from Figure 117 to focus on the stereotyping issue. The left-hand side of Figure 119 presents a *Spec* model while the right-hand side presents a *Design* model. The *Spec* model defines the *AntennaDevice* stereotyped as *DeviceComponent*, plus the *DeviceComponent* itself and two relationships. The *Design* model presents only an element called *AntennaDevice* and is not stereotyped. The question we have to ask is if the *Design* model is compliant with the *Spec* model. Using our current structural compliance rules and tools, the answer would be that elements and relationships from the *Spec* are missing in the *Design*.

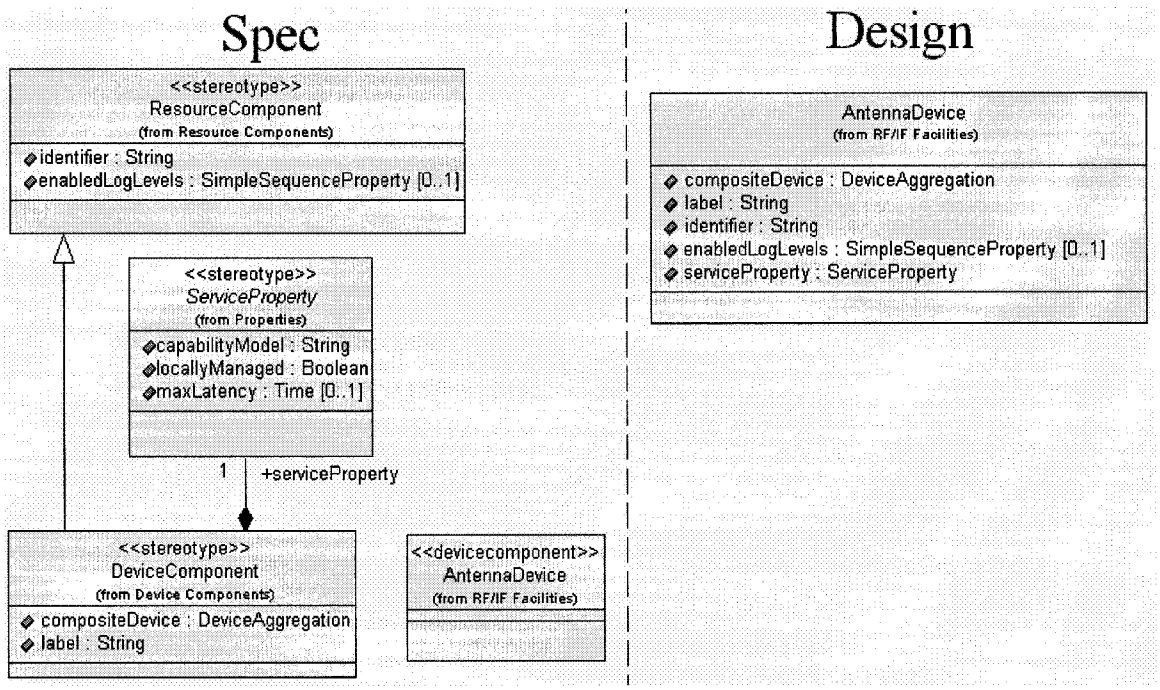


Figure 119. Stereotyping problem in structural compliance

At this point we would like to introduce a new concept that we call a *self-contained description* of an element. Such *self-contained description* of an element defines no relationships but considers properties added to the element by the element's relationships with other elements. Let us review the example in Figure 120. By the previous definition, the *self-contained description*s of ClassC in the three portions of the figure are the same. In the three cases *ClassC* contains 4 attributes. In Figure 120 (a), *ClassC* gets its four attributes by specializing *ClassA* and *ClassB*. In Figure 120 (b), *ClassB* defines two attributes, inherits two from *ClassA* and *ClassC* inherits the four. In Figure 120 (c), *ClassC* defines two attributes and inherits the other two from *ClassB*. Finally, in Figure 120 (d) *ClassC* defines itself four attributes. As the attributes in the three portions use the same names and the same primitive data types, we would say, that applying the *self-contained description* concept onto ClassC for the four different cases result in the same values.

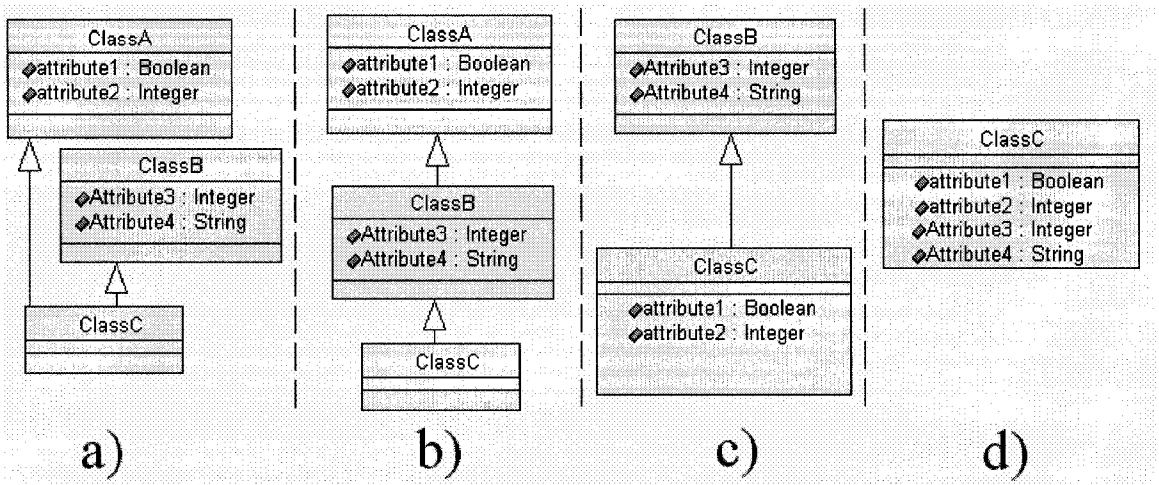


Figure 120. Self-contained description of an element (example)

Using *self-contained descriptions* may ease some challenges on structural compliance although it may bring some new challenges of its own. The example we show uses only generalization relationships but we also need to explore other cases were associations, compositions and aggregation relationships also play a role in the element's definition. Again we repeat, this topic is postponed for future consideration.

We conclude this section by finalizing our discussion of the SWRadio Profile and the use of prototypes in the SWRadio PIM. Using the concept of *self-contained descriptions*, we would say that the *Design* model in Figure 119 is compliant with the *Spec* model from the same figure. As the reader may infer, we believe a possible avenue to solve the stereotyping problem would use the concept of a *self-contained description.*

The next section gives a high level discussion of mappings that may be required if we were to apply our approach to the new SWRadio specification. The section concludes Chapter 4.

4.2.3 Additional mappings for the new SWRadio specification

In this section we discuss additional mappings that may be required to generate a semantically equivalent model (*tSpec*) for the new SWRadio specification. Although we do not define new mappings per se, we identify elements/relationships from the new

SWRadio *Spec* that have not been considered for mapping execution. For each type of element/relationship we discuss its context and outline a possible solution.

4.2.3.1 Stereotypes

When dealing with specification-specific elements like those defined in a domain-specific profile, a new mapping may be required. In our case, the OMG's Software-Based Communication (SBC) Domain Task Force (DTF) has defined a new Software Radio Specification that includes a SWRadio Profile. It has defined new SWRadio-specific elements with their own semantics. Such is the case of *EventChannels*, *commchannels*, *services*, *devicecomponents*, *radiomanagers, and many more.* Section *4.2.2* presents in detail our arguments that justify our call for a new mapping (we do not duplicate those arguments in this section, though). We invite the reader to review *4.2.2* for more information with respect of a possible *StereotypeMapping* definition.

4.2.3.2 Interface Dependencies

Within the *Spec* model, there are several dependency relationships from interfaces to data classes that have not been addressed by our current set of mappings. An example of such dependency relationships is shown in Figure 121. As discussed in section 4.1.1, in our current set of mappings interfaces are mapped into protocol classes in ROSE-RT, while classes are mapped into data classes or capsules. Semantics defined for ROSE-RT does not require the explicit definition of dependency relationships from a protocol class to other ROSE-RT elements. In other words, the dependency relationships shown in Figure 121 are not required to be present in the *tSpec* model shown in Figure 122.

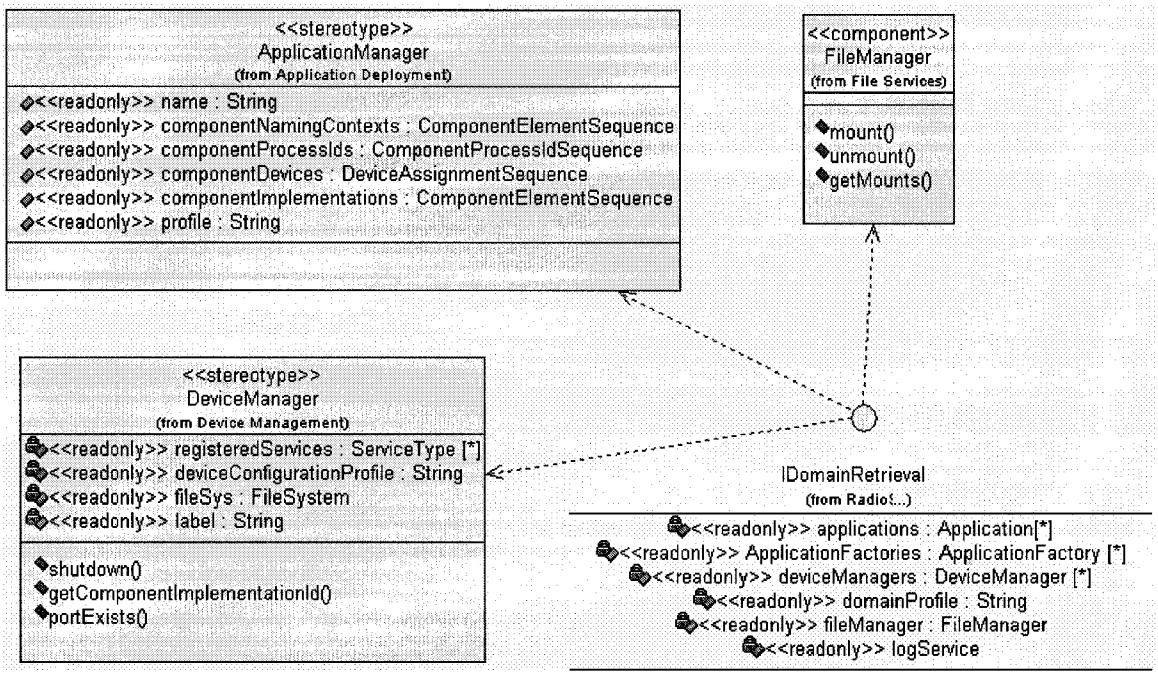


Figure 121. *Spec* interface dependencies to data classes

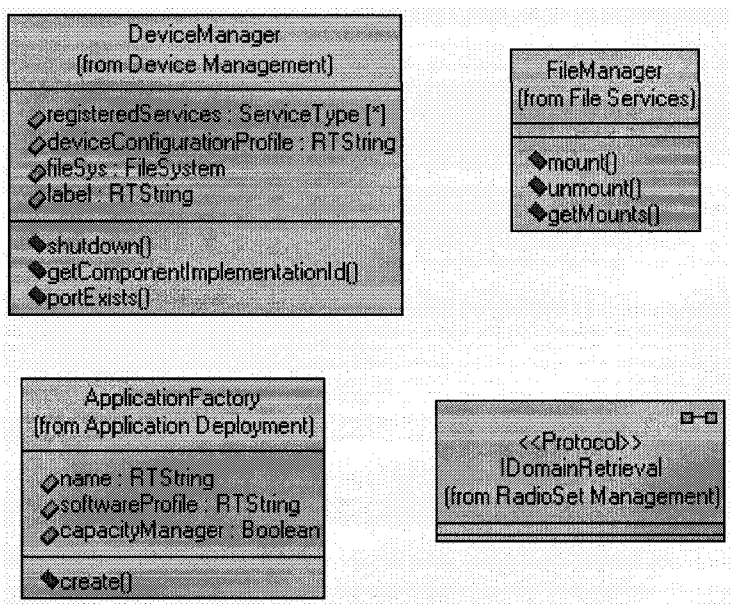


Figure 122. *tSpec* Protocol dependencies to data classes not required.

Although no action is required, the mapping could be used to mark the dependency relationships with a special tag that defines them as not required for mapping purposes. When such a label is placed on the dependency relationships, the 'number of times used'

variable can be incremented by one so they do not show as element not used for mapping purposes. It is possible also to not consider a new mapping, given the understanding that these kinds of dependency relationships will be viewed as not considered for mapping execution.

4.2.3.3 Missing elements in *Spec* model

Throughout the *Spec* model, a multitude of relationships contain references to non-existent elements in the *Spec* model. Figure 123 partially shows the output of a consistency checking for the new SWRadio Specification model. Our transformation tool requires both elements in a relationship to be present for a mapping to be applied. Having one of the elements missing, then the relationship is not mapped and reports it as a relationship that was not used as an originator of an element/relationship in the *tSpec* model.

```
10:00:36| Error: Unresolved reference from Use Case "Cell Search"
10:00:36|      to ClassItem with name (Unspecified)
10:00:36|      by Object "L1 Receiver".
10:00:36| Error: Unresolved reference to Item with name
10:00:36|      in AssociationView
10:00:36|      in Class Diagram: Communication Elements / Security Policy Associations
10:00:36|      Model for the association view cannot be resolved because there are no matches.
10:00:36|      Model for the association view cannot be resolved because there are no matches.
10:00:39| Error: Unresolved reference to Item with name Logical View::Facilities PIM::RF/IF
Facilities::AmplifierFacility
10:00:39|      in ClassView Logical View::Facilities PIM::RF/IF Facilities::AmplifierFacility
10:00:39|      in Class Diagram: IOFacility / Main
10:00:39| Error: Unresolved reference to Item with name Logical View::Facilities PIM::RF/IF
Facilities::FilterFacility
10:00:39|      in ClassView Logical View::Facilities PIM::RF/IF Facilities::FilterFacility
10:00:39|      in Class Diagram: IOFacility / Main
10:00:39| Error: Unresolved reference to Item with name Logical View::Facilities PIM::Common Layer
Facilities::Measurement Facilities::Imeasurement
10:00:39|      in ClassView Logical View::Facilities PIM::Common Layer Facilities::Measurement
Facilities::Imeasurement
10:00:39|      in Class Diagram: Quality of Service BB Element Definitions / IqualityofService Definition
```

Figure 123. Consistency checking on new SWRadio *Spec*

Although we cannot envision a new mapping for this situation, we do not close the door to consider the possibility that once those relationships have been fixed, a new mapping may be required for them. In other words, we cannot assure that all the fixed relationships will be mapped using one of the mappings already defined.

4.2.3.4 Parameterized Classes

Currently our set of mapping definitions does not include a mapping for parameterized classes. The mapping belongs to the semantically related element mappings group (see section 4.1.1), and would require mapping a parameterized class from the *Spec* model to be mapped into a parameterized class in the *tSpec* model. Within the *Spec* elements not used for mapping purposes, we also find parameters of parameterized classes not being mapped. Mapping parameters can be done within the same ParameterizedClassMapping or a new one.

4.2.3.5 Attributes in Interfaces

In the first version of the SWRadio interfaces define operations but not attributes. In the new specification, some interfaces also define attributes. In our current set of mappings an interface from the *Spec* is mapped into a protocol class from ROSE-RT. Protocol classes in ROSE-RT define incoming and outgoing signals but they cannot include attributes. A possible solution would be to map the interface attributes into each classifier that realizes the interface.

We conclude this section by recognizing that more mappings may be required as the ones suggested in this section are not meant to form an exhaustive list. Also, we remind the reader that with each new element being considered for mapping purposes, the nature of our approach provides us with clues about new mappings for the remaining elements/relationships that have not been considered for mapping purposes.

Chapter 5. Generalization of Contributions

In this chapter we discuss the generality of our contributions. We first address the generality of our mapping definitions if we were to generate *tSpec* models for specifications different that the SWRadio *Spec* we discussed in the previous chapter. We then present extensions to our mappings when trying to transform a UML 1.4 model into a UML 2.0 one that uses structured classes, ports and connectors.

Although our thesis research was done in a closed environment, we have many reasons to believe that our approach, and specially some of our contributions, can be generalized. We recognize that our mapping definitions are specific to the subset of UML elements that are used in the SWRadio PIM, and to the constructs that are used in our design models using the ROSE-RT notation. But such a selection was not a matter of a random choice. Our specification and design metamodels were carefully chosen thinking of the possibility of a more general application. Considering that two of the most commonly used UML diagrams are Class and Sequence diagrams [49], we chose a UML standard specification model that was built atop those two commonly used UML diagrams, namely the SWRadio PIM. On the design side, our selection of ROSE-RT was strongly motivated by the fact its core constructs (Capsules, Ports and Connectors) have been recently incorporated into version 2.0 of the UML specification.

Recall from Chapter 3 that in our approach, mapping definitions are used to transform a *Spec* model into a semantically equivalent *tSpec* model that uses the same language (metamodel) as the design models we want to verify for compliance. Mappings are established between elements of a source and a target metamodels.

5.1 Generalization of the current work on the two-step approach for model compliance verification

In this section we present general guidelines for applying our two-step approach for model compliance verification to other standard specifications. This section is built on top of the material already discussed in previous chapters and aims to reuse part of the work already done using the software radio specification. While analyzing each of the activities of the approach, we highlight the parts that may be reused and point to additional work when required. The discussion starts with a very general consideration on the use of any standard specification model. We then proceed with discussion of each of the activities of our approach while we narrow the scope of the analysis to the use of the Product Lifecycle Management (PLM) standard specification model [72]. We present examples of an attempt to generate a *tSpec* model for the PLM *Spec* model. As presented in Chapter 3, we first address the *Spec* to *tSpec* transformation and compliance activities and then discuss the *Design* to *tSpec* compliance activities.

5.1.1 *Spec* to *tSpec* Transformation and Compliance

In this section we discuss the generalization of our work related to the *Spec* to *tSpec* transformation and compliance. We use a subsection for each of the three activities that compose this first step.

5.1.1.1 Activity 1: Semantic definition and validation of mappings.

Within this activity the first task to work on is in the identification of the *Spec* and *Design* metamodels of the new standard specification and design models. Considering that we have already a SWRadio *Spec* and a ROSE-RT metamodels, the next step is the comparison of the new *Spec* metamodel with the SWRadio metamodel, as well as of the new *Design* metamodel with the ROSE-RT metamodel. Based on our experience, Table 46 presents different situations that can result from comparing our SWRadio *Spec* and ROSE-RT metamodels with other metamodels. The first column contains a sequential number. The second column describes the result of comparing the SWRadio *Spec* and a new *Spec* metamodel. We consider three possible values: 1) equal, 2) partially similar, and 3) different. The third column shows the result of comparing the ROSE-RT

metamodel with the new Design metamodel. We use the same three values as in the second column. The fourth column describes the action to be taken with respect to mapping definitions. Again we consider three possible actions: 1) do nothing, 2) update (add, change, remove) mapping definitions, and 3) define a new set of mappings. Finally the fifth column presents the reusability degree considering the mappings that have been defined already for the SWRadio *Spec* to ROSE-RT transformations. We present the reusability degree in values that range between very high and very low.

Table 46. Comparison between SWRadio *Spec* and ROSE-RT metamodels with other metamodels

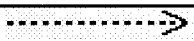
	SWRadio *Spec* and a new *Spec* metamodel	ROSE-RT and a new Design metamodel	Action	Reusability degree
1	Equal	Equal	Do nothing	Very High
2	Equal	Partially similar	Update (add, change, remove) mapping definitions	High/Low Depends on the similarities of the Design metamodels. The higher the similarity, the higher reusability
3	Equal	Different	Define a new set of mappings	Very low
4	Partially similar	Equal	Update (add, change, remove) mapping definitions	High/Low Depends on the similarities of the *Spec* metamodels. The higher the similarity, the higher reusability
5	Partially similar	Partially similar	Update (add, change, remove) mapping definitions	High/Low Depends on the similarities of the *Spec* and Design metamodels. The higher the similarities, the higher reusability
6	Partially similar	Different	Update (add, change, remove) mapping definitions	Very low
7	Different	Equal	Define a new set of mappings	Very low
8	Different	Partially similar	Define a new set of mappings	Very low
9	Different	Different	Define a new set of mappings	Very low

From Table 46, very high reusability comes when the specification and design metamodels are the same as in our case study. Reusability declines when differences are found between metamodels, and becomes very low when completely different metamodels (either *Spec* or Design) are used. As both our SWRadio *Spec* and ROSE-RT metamodels are UML-based, we believe that reusability of our mapping definitions with

UML-based specifications is located in the high to very high range. To apply our two step approach for model compliance verification with non-UML specifications most probably will require to start from zero the activity of defining and validating mapping definitions. In the remaining of the section we further discuss the generalization of our work considering three other UML-based standard specifications.

Table 47 presents elements and relationships used in the definition of the SWRadio PIM as well as three other different OMG standard specifications. The first column illustrates the graphical icon used for each element/relationship. The second column contains the name of the element/relationship. The third column contains a check mark if the element is used in the definition of the Software Radio Components Specification (SWRadio) [73]. Columns four through six contain a check mark if the element is used in the definition of the OMG's Data Distribution Service (DDS) [66], Super Distributed Object (SDO) [74] and Product Lifecycle Management (PLM) [72] specifications respectively.

Table 47. Elements and relatiomnships used in four OMG specifications

Graphical Icon	Model Element	SWRadio [73]	DDS [66]	SDO [74]	PLM [72]
	Class (*Spec*) Data Class (Design)	✓	✓	✓	✓
	Interface	✓	✓	✓	✓
	Package	✓	✓	✓	✓
N/A	Attribute	✓	✓	✓	✓
N/A	Operation	✓	✓	✓	✓
N/A	Parameter	✓	✓	✓	✓
	Lifeline	✓	✓	✓	✓
	Message	✓	✓	✓	✓
	Generalization	✓	✓	✓	✓
	Realization	✓	✓		✓
	Composition	✓	✓	✓	✓
	Dependency	✓			✓
	Aggregation	✓	✓		✓
	Association	✓	✓	✓	✓

From Table 47 we observe that the elements and relationships used in the SWRadio specification [73], on which our mapping definitions are based, constitute a super set of the elements used in the other three specifications. As we have defined in Chapter 4 a complete set of mappings for all the elements in the SWRadio *Spec* metamodel, we believe that *tSpec* models that use the ROSE-RT metamodel can be generated for the other three specifications with a very high degree of reusability of our mapping definitions. In the case of additional constraints attached to elements in the other specifications then modifications to our mapping definitions may be required.

In the next section we discuss activity two of our approach for model compliance verification if it were to be applied to different UML-based standard specifications.

5.1.1.2 Activity 2: Correct implementation of mappings

This activity defines the required tasks to verify that mapping definitions have been implemented correctly. Considering a UML-based standard specification context and

designs written using ROSE-RT constructs as design elements, this second activity needs only to be carried out for new mapping definitions, or if the expected results for existing mapping definitions change. In case additions/modifications were needed to our mapping definitions, this activity would be carried out as defined in section 3.1.2.

Following the comparison analysis presented in Table 47, and considering ROSE-RT as the design language, no further actions are needed for this activity at this moment. The next section addresses the third activity of our approach if we were to verify completeness of mapping definitions with different standard specifications.

5.1.1.3 Activity 3: Complete Definition of Mappings

This activity defines the tasks that verify that all elements defined in the *Spec* model have been used as originators of elements in the *tSpec* model. Following the same argument line as in the two previous sections, we believe that a complete set of mappings would have been defined if we were to generate a *tSpec* model for any of the other three standard specification models included in Table 47. Although we are using the same design metamodel, and also that mappings have been defined, tested and verified, this activity is still required. In the rest of this section we present an example of how the execution of this activity can lead us to consider the addition/modification of mapping definitions. We discuss the generation of a *tSpec* model using the OMG's Product Lifecycle Management (PLM) PIM as *Spec* model. The automatically generated PLM *tSpec* model can be used to check model compliance of PLM designs written using ROSE-RT.

After identification of the *Spec* and *Design* metamodels, we compare the PLM *Spec* metamodel with the SWRadio *Spec* metamodel looking to reuse mapping definitions. As shown in Table 47, the type of elements used in the PLM *Spec* are the same type of elements used in the definition of the SWRadio PIM, for which, we repeat, mappings have been defined, implemented and tested. At this point we believe then that there is no need for additional mappings to be implemented. We proceed then with the execution of our transformation tool using the PLM specification as our *Spec* model

The PLM specification is defined using 15 packages, which contain 194 classes and 32 interfaces. Classes defined in the PLM specification contain 184 attributes. Relationships in the PLM specification are composed of 186 simple associations, 306 compositions, 334 realizations, 2 dependencies and 185 generalizations. We run our prototype tool and obtain a *tSpec* model from the PLM *Spec* model. After successful execution of our transformation tool the third activity in our approach requires us to verify that all elements and relationships from the PLM *Spec* model are used as originators of elements in the *tSpec* model. In the case of PLM *Spec* elements, the results show that all elements are used as required. In the case of PLM relationships, they have not. The first execution shows that several relationships between elements in the PLM *Spec* model were not mapped into the *tSpec* model. In the following paragraphs, we present small examples of why some of those relationships were not mapped. We also outline possible solutions.

Of the original 1,013 relationships defined in the PLM *Spec* model, 379 were not used as originators of elements/relationships in the *tSpec* model. We found three possible causes. The first cause for some of the PLM *Spec* relationships not being mapped into the PLM *tSpec* model relates to interfaces, protocols and the communication mechanisms defined by the design metamodel (ROSE-RT). Figure 124 presents a class diagram from the PLM *Spec* that shows elements and relationships from the PLM configuration management package. The diagram shows classes (*Product_class, Class_condition_association, Class_specification_association, Specification_expression, Specification_inclusion* and *Specification*), an interface (*Specification_oper_and_select*), and composition, simple associations and realization relationships between elements in the diagram.

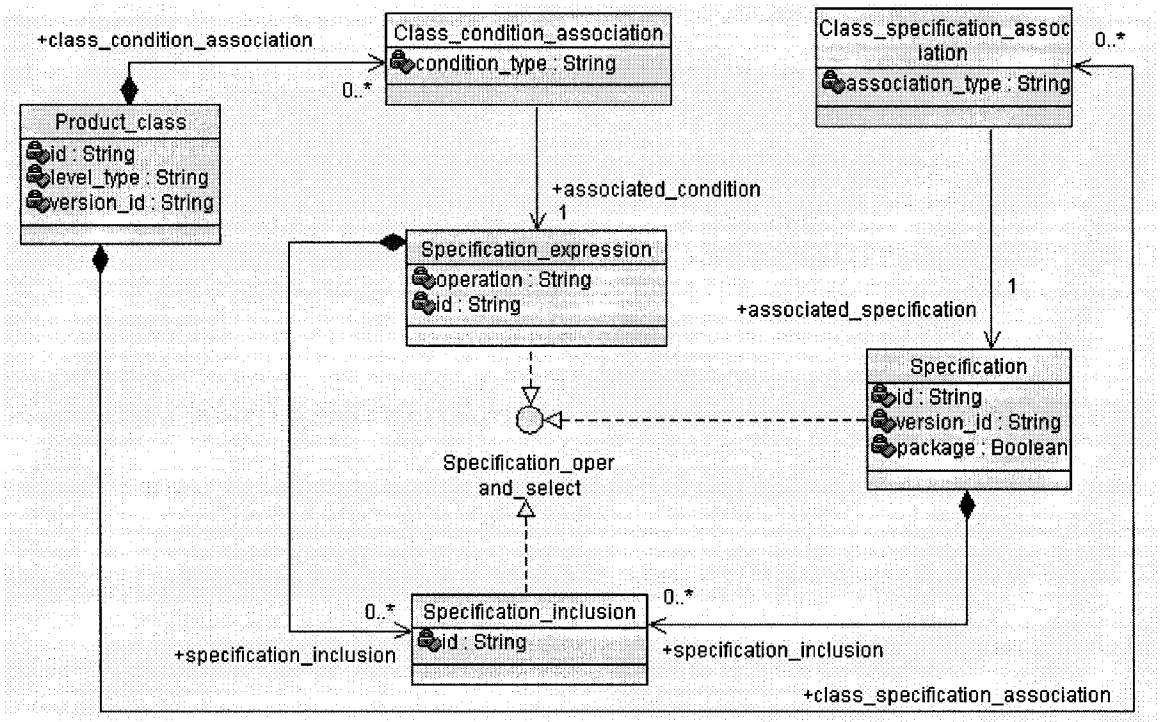


Figure 124. PLM *Spec* Configuration Management class diagram (for interface realization example)

An attempt to transform the PLM *Spec* model into a semantically equivalent PLM *tSpec* model produces some relationships from the PLM *Spec* model not being mapped into the *tSpec* model. Table 48 shows partial results (only those concerned with our example) that indicate that three realization relationships were not mapped into the *tSpec* model (shown in Figure 125).

Table 48. Partial PLM *Spec* elements used while verifying completeness of mapping deffinitions

#	Element Type	Element Name	Number of times used
1	Realize	Specification_operand_select_To_Specification	0
2	Realize	Specification_operand_select_To_Specification_expression	0
3	Realize	Specification_operand_select_To_Specification_inclusion	0

Figure 125 shows the *tSpec* model that corresponds to the fragment of the *Spec* model shown in Figure 124.

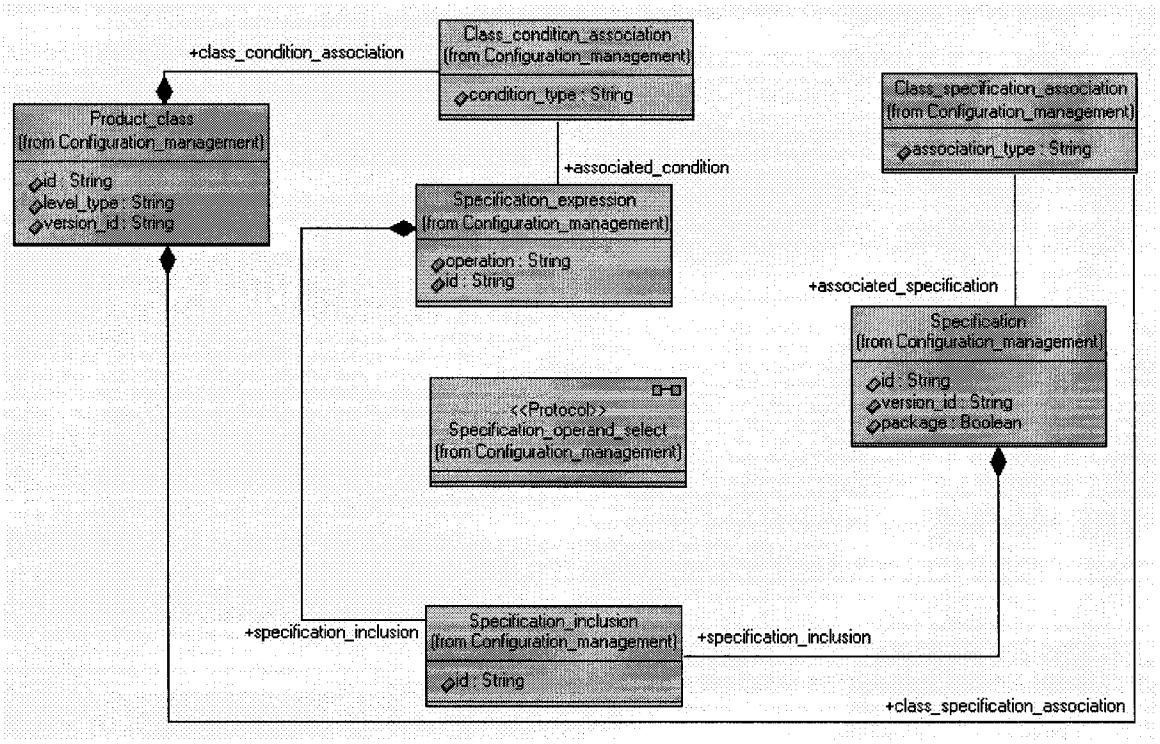


Figure 125. PLM *tSpec* Configuration Management class diagram with missing relationships (for interface realization example)

The tool successfully transformed classes and attributes from the PLM *Spec* model into equivalent elements in the *tSpec* model. It also mapped the *Specification_operand_select* interface into a protocol in the *tSpec* model. Composition and simple association relationships were also mapped. The relationships that were not mapped in this example correspond to realization relationships of an interface from three different classes on the *Spec* model. The reason the realization relationships is not being mapped into the *tSpec* model does not directly translate into an incorrect implementation, but rather incomplete information that indirectly involves the *Design* metamodel. In ROSE-RT, relationships with a protocol class are always of type composition, and are only defined between a capsule (an active class) and a protocol. The result of such composition relationships is the definition of ports that enable communication between capsules. In our example, we have not defined any active classes related to the PLM *Spec* model. It is required to do so if we are to successfully map those realization relationships (under the current mapping definitions). We run our transformation tool again, but this time using transformation input as shown in the XML file in Figure 126,which defines

that *Specification*, *Specification_inclusion* and *Specification_expression* are PLM *Spec* active classes to be mapped as capsules in the PLM *tSpec* model

```
<?xml version="1.0"?>
<transformationInput>
  <typeCorrespondenceDefinition>
  </typeCorrespondenceDefinition>
  <activeClassDefinition>
    <activeClass>PLM_services::Configuration_management::Specification</activeClass>
    <activeClass>PLM_services::Configuration_management::Specification_inclusion</activeClass>
    <activeClass>PLM_services::Configuration_management::Specification_expression</activeClass>
  </activeClassDefinition>
  </associationDefinition>
  <operationDefinition>
  </operationDefinition>
</transformationInput>
```

Figure 126. PLM transformation Input (for interface realization example).

Figure 127 shows a *tSpec* model that considers Figure 126 as additional transformation input. The new *tSpec* model now shows composition relationships from the capsules to the protocol class.

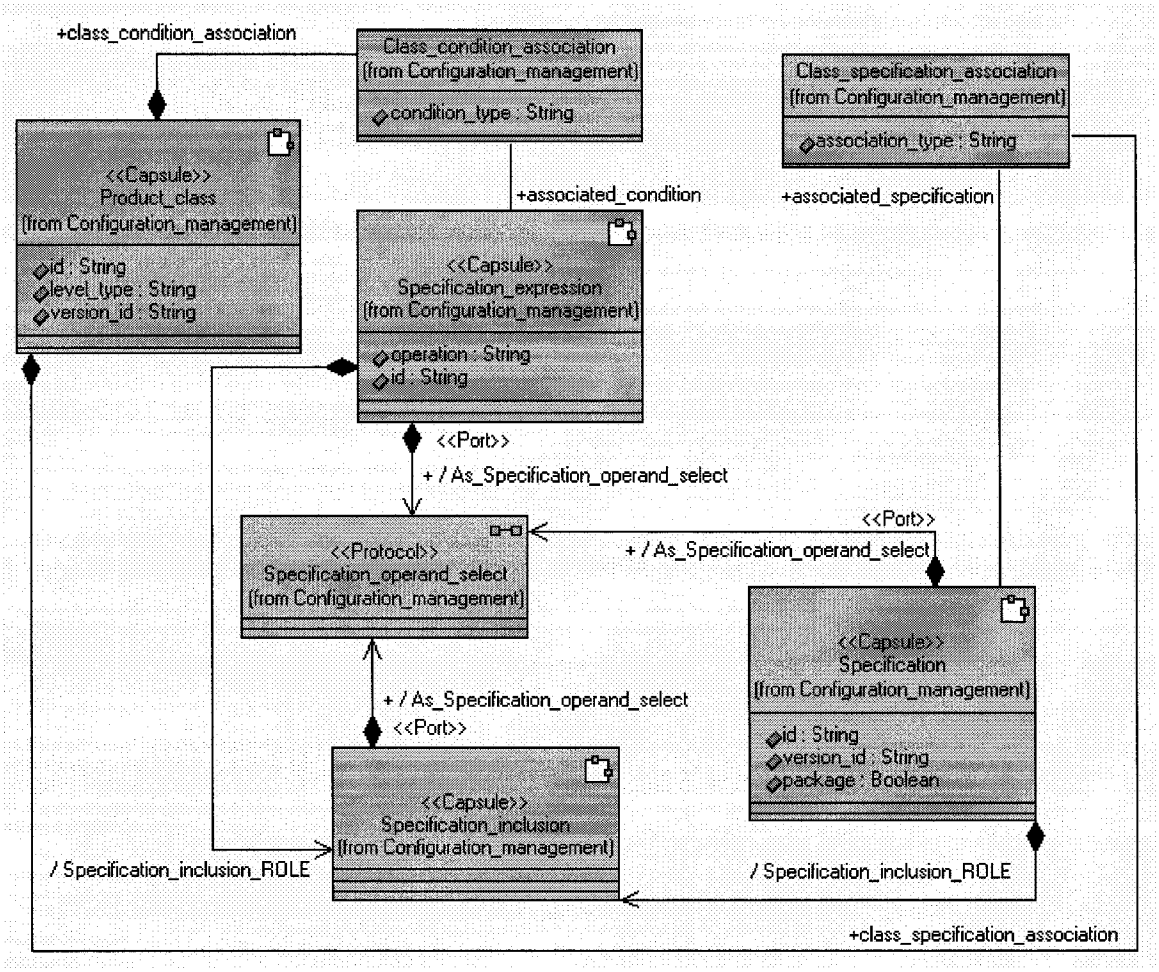


Figure 127. PLM *tSpec* Configuration Management class diagram with port definitions (for interface realization example)

Additional information in which some active classes were defined took care of the first cause of relationships from the PLM *Spec* model not being mapped into the *tSpec* model. The next cause for discussion also involves constraints in the design metamodel. ROSE-RT only allows generalization relationships between the same types of elements. ROSE-RT does not permit a data class to be a specialization of a capsule, nor a capsule to specialize a data class. When we first dealt with realization relationships not being mapped, we defined several classes from the *Spec* model as active classes to be mapped as capsules in the *tSpec* model. Now, we have a problem of attempting to generalize/specialize different types of classifiers. Figure 128 presents a PLM *Spec* diagram where generalization relationships are defined.

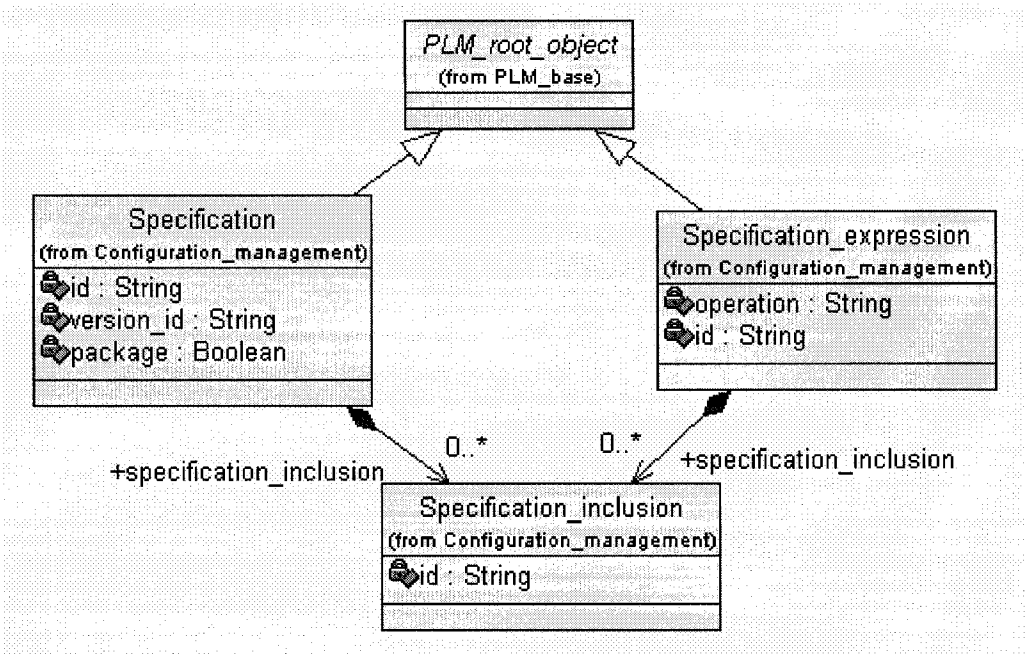


Figure 128. PLM *Spec* Configuration Management class diagram (for generalization example)

An attempt to transform the PLM *Spec* model from Figure 128 into a semantically equivalent PLM *tSpec* model results in some relationships from the PLM *Spec* model not being mapped into the *tSpec* model. Table 49 shows partial results (only those concerned with our example) that indicate that two generalization relationships from the PLM *Spec* model were not mapped into the *tSpec* model (shown in Figure 129).

Table 49. Partial PLM *Spec* elements used while verifying completeness of mapping deffinitions

#	Element Type	Element Name	Number of times used
1	Generalization	PLM_root_object_To_Specification	0
2	Generalization	PLM_root_object_To_Specification_expression	0

Figure 129 shows the *tSpec* model that corresponds to the fragment of the *Spec* model shown in Figure 128.

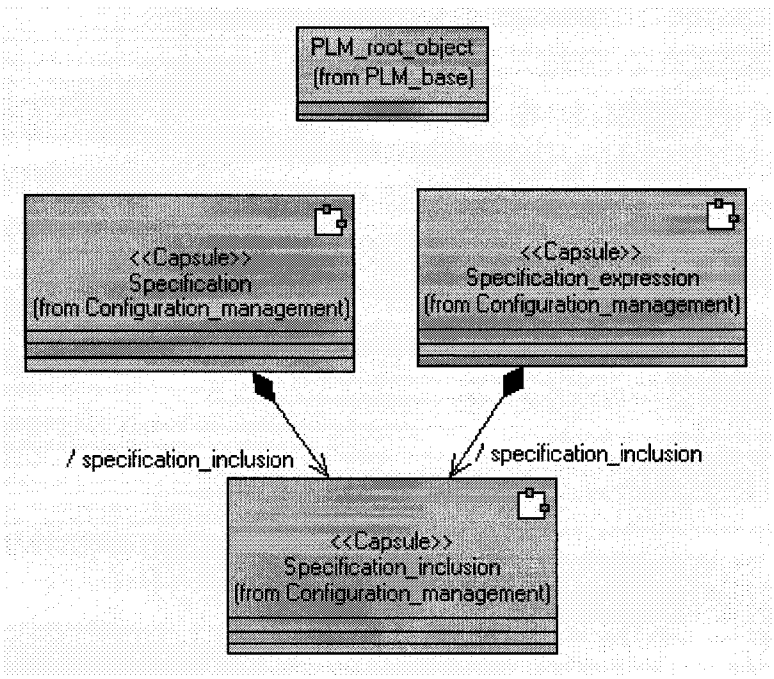


Figure 129. PLM *tSpec* Configuration Management class diagram with missing relationships (for generalization example)

This second cause for missing some relationships in the *tSpec* model can also be overcome by providing additional input that defines that the PLM_root_object from the PLM *Spec* model is also an active class, as we show in Figure 130.

```
<?xml version="1.0"?>
<transformationInput>
  <typeCorrespondenceDefinition>
  </typeCorrespondenceDefinition>
  <activeClassDefinition>
    <activeClass>PLM_services::PLM_base::PLM_root_object</activeClass>
    <activeClass>PLM_services::Configuration_management::Specification</activeClass>
    <activeClass>PLM_services::Configuration_management::Specification_inclusion</activeClass>
    <activeClass>PLM_services::Configuration_management::Specification_expression</activeClass>
  </activeClassDefinition>
  </associationDefinition>
  <operationDefinition>
  </operationDefinition>
</transformationInput>
```

Figure 130. PLM transformation Input (for interface realization example).

Execution of our transformation tool with the new information as provided in Figure 130 produces the *tSpec* model shown in Figure 131 where all the elements and

relationships from the diagram in Figure 129 were used as originators of elements/ relationships shown in the partial *tSpec* model shown in Figure 131.

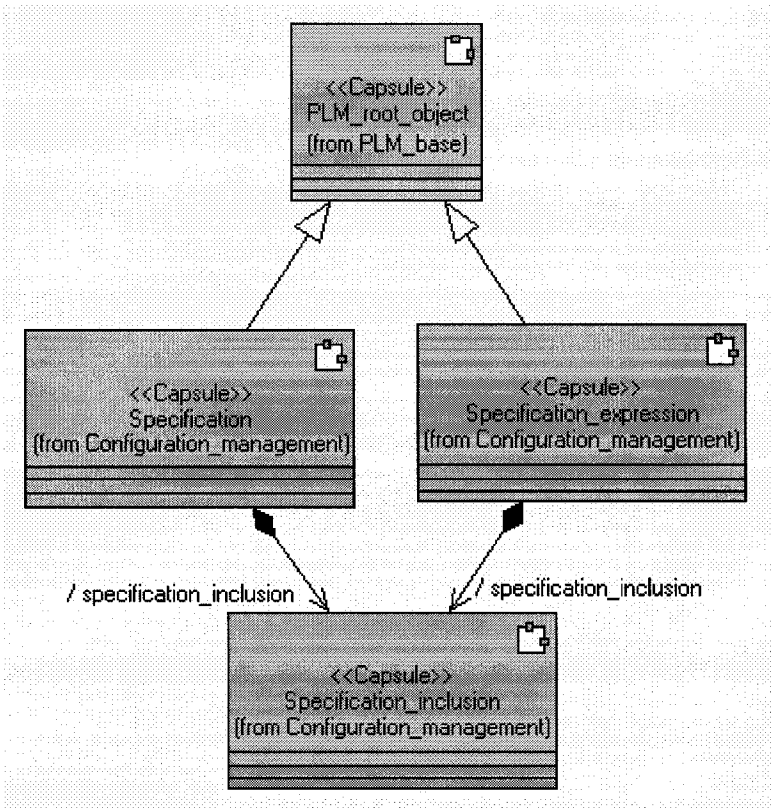


Figure 131. PLM *tSpec* Configuration Management class diagram with generalization relationships (for generalization example)

After we have addressed the interface realization and generalization problems, there are still 46 simple association relationships from the PLM *Spec* model that have not been considered as originators of elements/relationships in the *tSpec* model. These associations represent the third and final cause in which relationships from the PLM *Spec* model were not used as originators of elements/relationships of the PLM *tSpec* model. The solution is not simple and requires additional work. We further discuss it and present an example in the paragraphs below.

Figure 132 presents PLM elements and relationships. The reading of the diagram is as follows: A *Specification* of a product may contain 0 or many *Specification_inclusions*

(composition relationship between *Specification* and *Specification_inclusion*). A *Specification_expression* may contain 0 or many *Specification_inclusions* (composition relationship between *Specification_expression* and *Specification_inclusion*). A *Specification_inclusion* holds a reference (association between *Specification_inclusion* and interface *Specification_operand_select*) to an element that might be of type *Specification* or of type *Specification_expression* (illustrated by means of *Specification* and *Specification_expression* realizing the *Specification_operand_select* interface). The latter is not clear from the diagram, but described in natural language in the PLM specification document where the interface *Specification_operand_select* is defined as: "interface defined to provide a placeholder for the following classes: *Specification_expression* or *Specification*" [72].

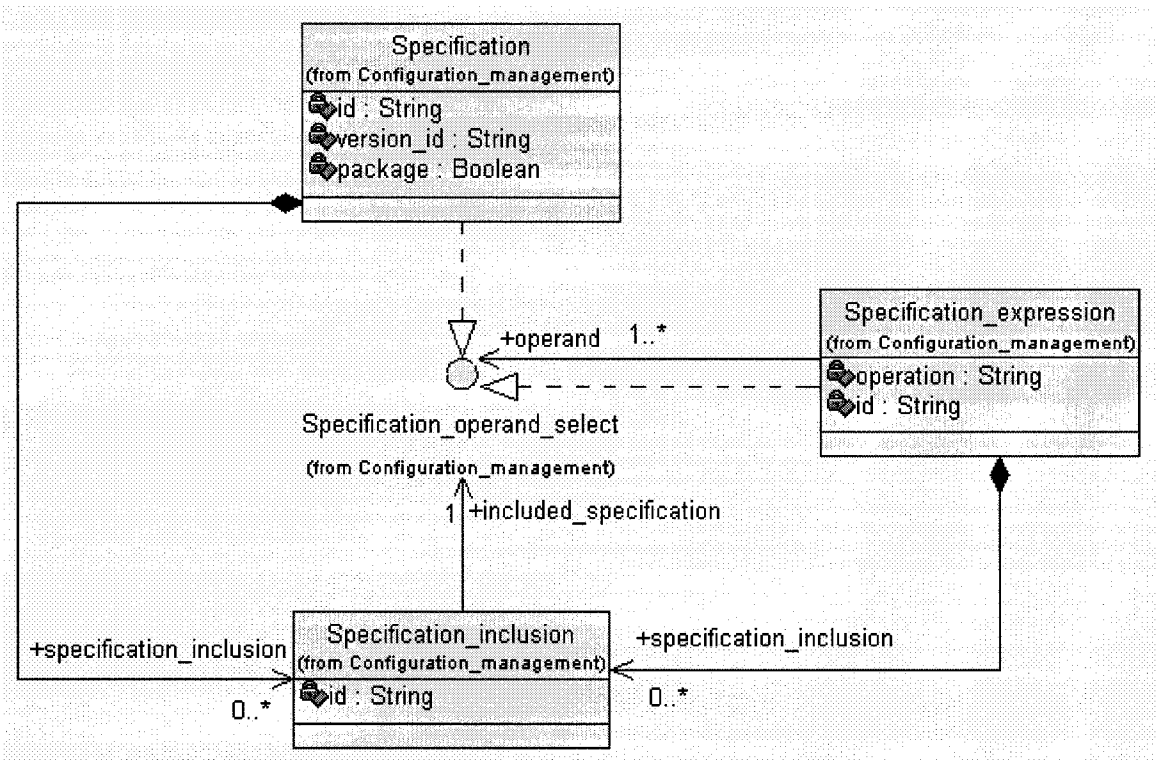


Figure 132. PLM *Spec* Configuration Management class diagram (for association of a class with an empty interface)

With such semantics attached to the model, our current mapping definitions are not capable of handling associations with 'empty interfaces used as placeholders'. If we were

to apply our approach for compliance verification of PLM Designs, it would require that we modify/add some mapping definitions to those of section 4.1.1. We do not modify nor define new mappings, as the objective of this chapter is to illustrate the generality of work already done. We consider that we have given the reader sufficient basis to believe that there is a very high degree of reusability to our work. At the same time that we do recognize that additional work is required when working with different standard specifications. Still the amount of work of the first three activities has been reduced to the addition of one or two mappings, perhaps the modification of a couple of existing ones, and the verification of correctness of the implementation of those mappings. We reiterate, we believe that there is a high degree of reusability of our mapping definitions.

The next section addresses the generality of the second step of our approach for model compliance verification.

5.1.2 Design to *Spec* Compliance

This section discusses the generalization of our work with respect of the activities four and five of our approach. We use individual subsections for each activity.

5.1.2.1 Activity 4: Structural Compliance

In this section we address the generality of our work with respect of structural compliance of a *Design* model with respect of a *tSpec* model.

Our proposal for structural compliance considers different aspects (see section 3.2.1) in which we compare the correspondence of two models. We have also built a tool to carry on such comparison. The tool has been implemented to read and parse the two models, and execute comparisons according to the aspects discussed in section 3.2.1. Considering the generality of our structural compliance proposal, there are two lines of work to pursue. The first line considers the reading of the models, while the second contemplates defining new comparison rules and even modifying existing ones.

In terms of reading the models, our tool is prepared to read models written using ROSE-RT. This situation does not affect the compliance verification part though. In other words, if the models (*tSpec* and *Design*) were written in a different language, the reading/parsing would be required to change, but not the model comparison.

In terms of the comparison rules and the structural aspects that we have considered in this thesis, we recognize that additional aspects (and hence additional comparison rules) may be required to address the needs of each compliance verification effort. We do not claim to have defined a complete set of comparison rules, nor to have considered all possible aspects when addressing the structural compliance of two models. On the other hand, if the *tSpec* and *Design* models use the ROSE-RT metamodel and no additional structural aspects (e.g., stereotypes) are to be considered, our work is highly reusable as it is independent of the standard specification we want to verify. In other words, if we have ROSE-RT *tSpec* and *Design* models, our current work can be reused in its actual state to verify for structural compliance regardless of the standard specification it is aimed to verify.

The next section addresses the generalization of our work with respect of behavioral compliance of two models.

5.1.2.2 Activity 4: Behavioral Compliance

In this section we address the generalization of our work with respect of behavioral compliance of a *Design* model with respect of a *tSpec* model.

Our proposal for behavioral compliance considers different challenges than the ones for structural compliance. First, there is the formalism used to specify the mandatory behavioral definition of the specification model. Then there is tool support that exists for the verification of such behavior. Finally, we need to contemplate the transformation of the behavioral definition into the same language used by the design model.

With respect to the behavioral formalism, in section 1.5 we presented our arguments for the selection and use of sequence diagrams in our approach. We reiterate that whereas we do not consider the use of sequence diagrams to be mandatory, we do consider activity 5 (in which we verify for behavioral compliance) to be mandatory within our approach. We consider that the essence of the activity is more important than the implementation choice. If the specification's mandatory behavioral definition is described in terms of state machines, then the implementation of activity 5 will need to be modified to meet the new requirements. In this thesis, we present only a specific instance of how such activity may be carried on using sequence diagrams as the formalism used for behavioral definition.

With respect to the tool we used to verify behavioral compliance, we did not propose any new tool or mechanism, because behavioral compliance must not only consider how to compare two behavioral definitions, but also have one of these behavioral definitions tightly coupled to the actual execution of a model. Although an interesting research topic, we considered implementing behavioral compliance to be beyond the scope of this thesis. We opted instead for available off the shelf MDD tools that already implement such tasks. The way they do it, and whether they fully or partially address what we call behavioral compliance is also beyond the scope of this work. With respect to the choice of such a tool, in section 2.1 we presented our arguments for our selection. We also reiterate that our decision to use Rational Rose RealTime does not prevent the use of Objecteering Tests for Java/EJB © [61], I-Logix Raphsody © [31] or Telelogic Tau © [100] as potential others tools to implement our approach.

Finally, there is the consideration of the language used in the design. This aspect is more concerned with the behavioral formalism used in the *Spec* and *Design* models than the ROSE-RT metamodel itself. Let us explain. In our case study we presented a Specification model that uses sequence diagrams as the behavioral formalism. Those sequence diagrams not only represent interaction among entities of the SWRadio *Spec*, but also define the communication approach of such interaction. Recall from section 2.1.5 the implications of the use of a procedure call communication approach in the *Spec*

model, while using a signal based communication approach for the Design model. Such selection of the communication approaches implies mapping definitions to transform procedure call based sequence diagrams from the *Spec* into signal-based sequence diagrams in the *tSpec* model. The latter implies, for instance, two messages being interchanged in the *tSpec* model for every procedure call in the *Spec* model (see section 2.1.5). In terms of generalization of the work already done, even if we kept using sequence diagrams, a change in the communication approach in any of the *Spec* or *Design* models implies the modification of our mapping definitions that maps *Spec* sequence diagrams into the *tSpec* model.

5.2 Gap challenges on the generalization of the two step approach for compliance verification

We focus the work of this section on the gap that exists between the *Spec*, *tSpec* and *Design* models we use. We discuss how an increase/decrease of such gaps might affect the application of our approach. Throughout the section, we refer to two different types of gaps (Δ's in the figures): the gap (Δ_1) that exists between *Spec* and *Design* metamodels, and the gap (Δ_2) that exists between the *tSpec* and *Design* models. In the case of Δ_1, we remind the reader that our *Spec* and *Design* metamodels are only composed of metaclasses that are used in defining the *Spec* and *Design* models. In other words, recall from Chapter 4 that our *Spec* metamodel is particular to the elements of UML used in defining the *Spec* model, and not the whole UML specification.

Figure 133 presents three examples where the gaps (deltas in the figure) between models are increased/decreased. The first example presents a base case for comparison with the examples two and three. In the second example, the gap (represented by Δ_1 in the figure) between *Spec* and *tSpec* models is reduced (when compared with example 1), meaning that the metamodels used by the *Spec* and *Design* models are closer than the ones in example 1. At the same time, example 2 presents a wider gap between the *tSpec* and *Design* models (represented by Δ_2 in the figure)[14]. In the third example of Figure 133

[14] Although a reduction of the Δ_1 gap might suggest an increase in the Δ_2 gap (as implied in Figure 133) we do not acknowledge such correlation. Further research is needed to derive a conclusion in such a matter.

we present a case where the gap Δ_1 is increased (again, when compared with the example 1) whereas the gap Δ_2 is decreased.

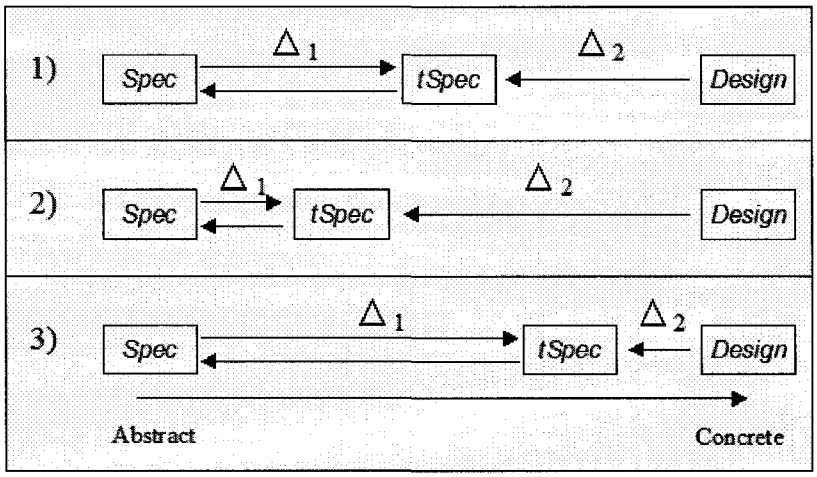


Figure 133. Deltas between *Spec*, *tSpec* and *Design* models

We use the following two subsections to further discuss the Δ_1 and gap Δ_2 gaps from Figure 133.

5.2.1 Gap between Spec and Design metamodels: Δ_1

In this section we further discuss the impact of reducing/increasing the gap between *Spec* and *Design* metamodels. Recall from Chapter 3 that the *tSpec* is expressed using the same language (metamodel) as the *Design* model. Also recall from Chapter 3 that the *tSpec* model is automatically created using a set of mappings that transform elements from the *Spec* metamodels into element(s) of the *Design* metamodel. The result of applying such mappings is the creation of the *tSpec* model that is composed of elements created upon elements from the *Spec* model. In general terms, the more distant the *Spec* and *Design* metamodels are, the more complex the definitions of those mappings.

The mappings we defined in our case study in Chapter 4 were based on two UML-based metamodels. Although using different constructs and communication mechanisms, both *Spec* and *Design* metamodels had the same core metamodel structure: the UML metamodel. Figure 134 presents a hierarchy where our *Spec* and *Design* metamodels are

shown in the bottom-left area of the figure. To bridge the gap between the *SWRadio Spec* and *SWRadio Design* metamodels it was necessary to consider the UML metamodel itself. If we were to use some other UML-based *Design* language (third box from left to right in the figure), most probably the work would be similar as the one developed in Chapter 4, as it is also necessary to go through the UML metamodel. If we were to define mappings between two CWM-based metamodels, we suggest that a similar level of complexity would be required. In other words, the fact that both metamodels are based on the same parent metamodel requires nothing further than working with the parent metamodel itself. If on the contrary we were to verify compliance of a CWM-based *Design* against a UML-based *Spec*, the task not only involves working with two different parent metamodels, but also addressing a common meta-metamodel (MOF in the current example). In the latter, two levels-up from the current *Spec* and *Design* metamodels would be required in order to bridge the metamodel gap in order to define mappings. Finally, there is the question of checking compliance of a non-MOF *Design* (empty dotted box at the right of Figure 134) against a UML based *Spec* model as our SWRadio PIM. As the figure suggests, it would be required to go three levels up to a non-existing (at least not at the moment of printing this thesis) parent meta-meta-metamodel that would allow us to define such mappings.

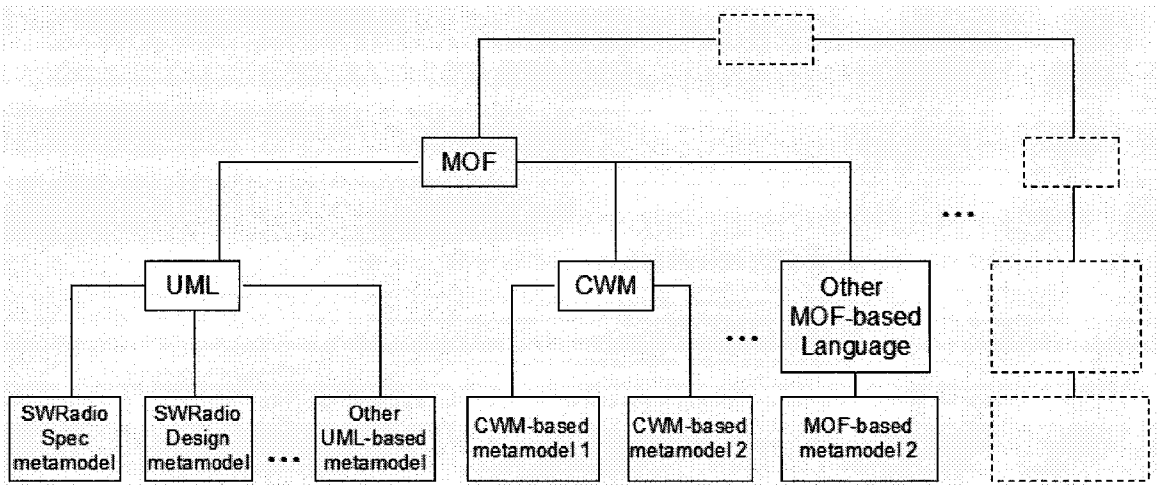


Figure 134. MOF-based languages

Through this thesis we have only worked with UML-based models (SWRadio and ROSE-RT in the case study of Chapter 4 and the PLM specification in section 5.1). We have worked at the bottom of the hierarchy and further research is needed to investigate the feasibility of verifying compliance of non-UML Designs against an UML-base specification. We end this section by pointing the reader to the work of Jean-Marie Favre [23] that suggests a *Megamodel* as a possible candidate to fill the empty dotted box that occupies the top of the hierarchy on Figure 134.

5.2.2 Gap between tSpec and Design models: Δ_2

In this section we briefly discuss the challenge that increasing/reducing the gap (Δ_2 from Figure 133) between *tSpec* and *Design* models implies to the generalization of our approach. Recall from Chapter 3 that we verify compliance of a Design model against an automatically generated *tSpec* model. The language gap Δ_1 between *Spec* and *Design* models has been bridged during generation of the *tSpec* model. Thus compliance is performed between models that use the same modeling language.

Recall also from Chapter 3 that we perform structural and behavioral compliance as separate activities. In the case of structural compliance, the gap between *tSpec* and *Design* models increases/decreases as we consider (or not) some of the challenges (and perhaps even others not addressed in this thesis) described in Chapter 3. In terms of behavioral compliance, recall also that behavioral compliance is execution-based and analyzes the capability of a *Design* model to send/receive messages as defined in *tSpec* sequence diagrams. Under such characteristics, behavioral compliance is not directly impacted by an increase/decrease on the gap between *tSpec* and *Design* models. We use the remaining of this section to address the challenge of a generalization of the structural compliance activities of our approach.

Perhaps the most important issue in considering generalization of the structural compliance activities relies on a vague definition of compliance. Recall from Chapter 3 that while for some a *Design* model might be compliant with a Spec model, for others it might not. That is why we introduced compliance rules to allow for variability in *Design*

models and still consider a model to be compliant. In general terms, the more rules are fed into the process, the more complex verifying for structural compliance will be. The more rules, the greater the gap between a *tSpec* and a *Design* model. As we have mentioned before, we have not defined a complete catalogue on how to bridge all possible gaps, but attempted to provide examples on the more common cases, as well as recommended solutions for them. We believe that to consider an additional challenge that increases the gap between *tSpec* and *Design* models, can be incorporated into the approach following the same process we used to define the compliance rules we propose in Chapter 4.

There is though an aspect that we have not explored that can substantially increase the complexity of this process. This new aspect is similar to metamodel gap but now between *tSpec* and *Design* models. While it gives greater freedom on design choices, it considerably increases the complexity of structural compliance verification. We present an example below.

Recall from Chapter 4 the Interface Realization challenge. Through the case study we mapped interfaces from the *Spec* model into protocol classes in order to generate the *tSpec* model. We reproduce some figures below (see Figure 135 and Figure 136). Structural compliance was then to look for protocol classes in the *Design* model to match those of the *tSpec* model.

Suppose now for instance, that interfaces where also valid elements in the *Design* language. Suppose also that instead of mapping interfaces from the *Spec* model into protocol classes in the *tSpec* model, interfaces were copied as interfaces into the *tSpec* model. Consider finally that the *Design* model realizes interfaces from the *Spec* model in two different ways: as interface definitions or as protocol classes. Considering the previous assumptions, the tSpec for our example model would be similar to that of the Spec model on Figure 135, and not the one that appears in Figure 135 and Figure 136. The problem then lies in how to verify compliance of a *Design* such as the one we present in Figure 137.

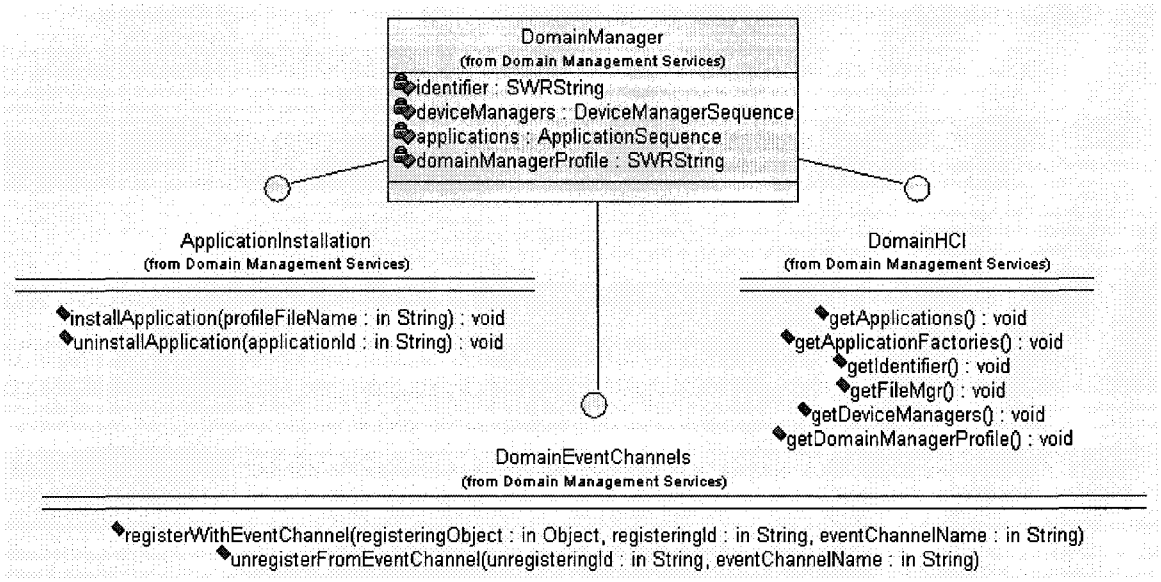


Figure 135. *Spec* DomainManager interfaces

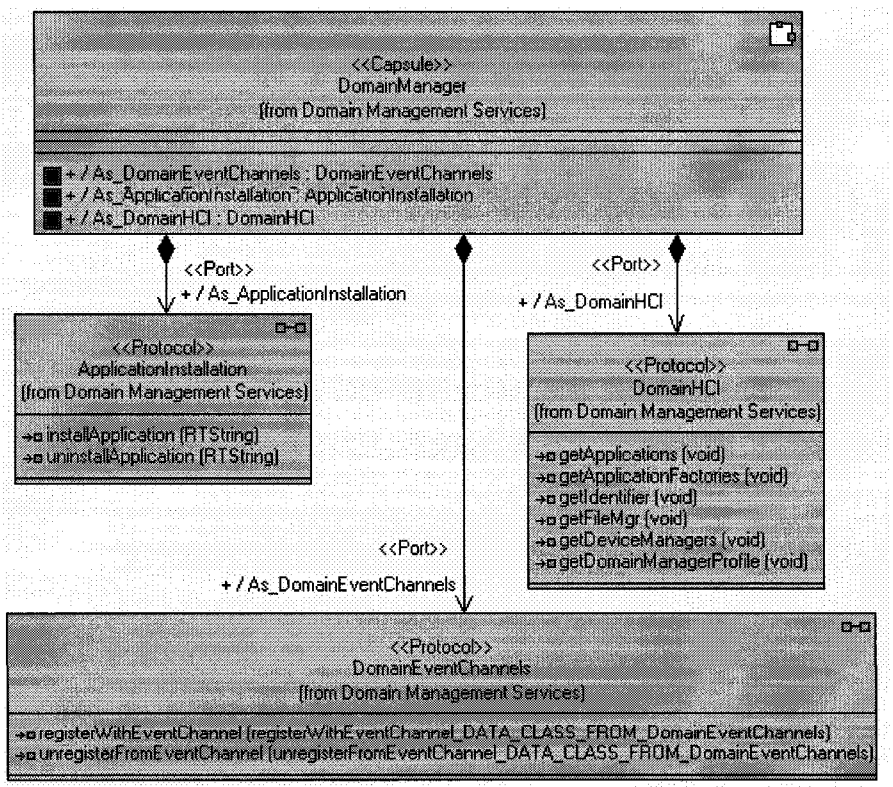


Figure 136. *tSpec* Domain Manager composition relationships to protocol classes

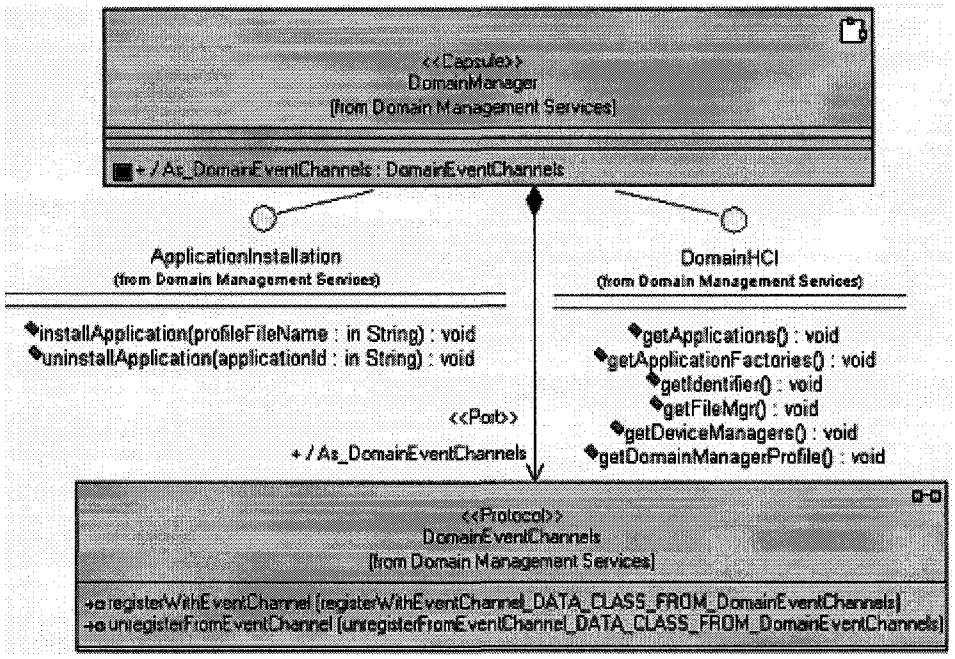


Figure 137. *Design* DomainManager model

Such added freedom on allowing different options (and constructs) to realize a single element from the *Spec* model, implies that although *tSpec* and *Design* models are written in the same language, we need to embed transformation mechanisms into our structural compliance process. We consider this topic part of our future research.

5.3 Additional Mappings towards UML 2.0

In this section we define additional mappings that extend our mappings in order to handle transformations into UML 2.0.

In terms of a general application of our mappings, we think of a UML 1.x to UML 2.0 transformation. In this section we discuss the required changes/additions to our mappings in order to achieve such a transformation.

As mentioned before, the core concepts of the ROSE-RT notation are Capsules, Protocol classes, Ports and Connectors. Three of them have a similar concept in UML

and thus only one requires additional work. Table 50 presents the equivalent concepts in both notations.

Table 50. ROSE-RT and UML 2.0 core concepts

ROSE-RT	UML 2.0
Capsule	Structured Class
Port	Port
Connector	Connector
Protocol Class	Establishment of a realization relationship between a port and required and provided interfaces.

The following mappings are in addition to the ones defined in section 4.1.1.

CapsuleToUML2Mapping (from UML 2.0 Mapping)

Maps a *ROSE-RT::Capsule* from the source model into a *UML2.0::StructuredClass* in the target model.

Context

ROSE-RT::Capsule **c**, UML2.0::Model **t**

Actions

```
\\ Create a new Structured Class using the same name of the Capsule
StructuredClass s1 = new StructuredClass (c.name)
t.add(s1)
```

Constraints

invoked from **ROSE-RT_to_UML2.0**

P1 \\ Source element is a capsule
c.oclIsTypeOf(Capsule)

PortToPortMapping (from UML 2.0 Mapping)

Maps a *ROSE-RT::Port* from the source model into a *UML2.0::Port* in the target model. Although the port concept is similar in both ROSE-RT and UML 2.0, the messages that can flow through them vary in their definition. In essence, while ports in ROSE-RT are capable of sending/receiving signals as defined in a ROSE-RT protocol class, ports in UML 2.0 are based on required and provided interfaces. In addition to creating a UML 2.0 port, this mapping requires to establish the realization relationship from the port to the required and provided interfaces.

Context

ROSE-RT::Port p1, UML2.0::Model **t**

Actions

```
\\ Create a new Port using the same name of the ROSE-RT port
Port p2 = new Port (p1.name)
\\ Establish the required interface for the port
p2.required->select (port.name+"_as_required");
\\ Establish the provided interface for the port
p2.provided->select (port.name+"_as_provided");
\\ add port to the model
t.add (p2);
```

Constraints

invoked from **ROSE-RT_to_UML2.0**

P1 \\ Source element is a port
c.oclIsTypeOf(Port)

ProtocolToInterfacesMapping (from UML 2.0 Mapping)

Maps a *ROSE-RT::Protocol* from the source model into *UML2.0::Interfaces* in the target model. The protocol concept does not exist in UML 2.0, but can be created by the combination of required and provided interfaces of a port.

Context

ROSE-RT::Protocol p1, UML2.0::Model **t**

Actions

```
\\ Create a new interface and label it as required.
Interface requiredInterface = new Interface (p1.name+"_as_required"))
\\ For all outgoing signals in the protocol class
for all (out signals in p1) {
\\ create an operation for every outgoing protocol signal
  Operation o = new Operation(signal.name);
\\ create a parameter to send data attached to the signal
  Parameter param1=new Parameter ("parameter");
\\ assign the type of the signal to the type of the parameter
  param1.type=signal.type
\\ add parameter to operation
  o.add(param1)
\\ add operation to interface
  requiredInterface.add(o)
}
\\ Create a new interface and label it as provided.
Interface providedInterface = new Interface (p1.name+"_as_provided"))
\\ For all incoming signals in the protocol class
for all (in signals in p1) {
\\ create an operation for every incoming protocol signal
  Operation o = new Operation(signal.name);
\\ create a parameter to send data attached to the signal
  Parameter param1=new Parameter ("parameter");
\\ assign the type of the signal to the type of the parameter
  param1.type=signal.type
\\ add parameter to operation
  o.add(param1)
\\ add operation to interface
  requiredInterface.add(o)
\\ add required interface to the target model
t.add (requiredInterface);
\\ add provided interface to the target model
t.add (providedInterface);
```

Constraints

invoked from **ROSE-RT_to_UML2.0**

P1 \\ Source element is a protocol class
c.oclIsTypeOf(ProtocolClass)

We conclude this section with a graphical example of how a ROSE-RT model (using capsules, protocol classes and ports) is transformed into a semantically equivalent model expressed using UML 2.0.

Figure 138 presents an example of a port definition in ROSE-RT. On the left side of the figure, a capsule *ElementA* contains a port *PortC*. The containment is expressed through a composition relationship from the capsule to the protocol class *ProtocolB*, which defines two incoming signals (signal1 and signal2 of type int and bool respectively). *ProtocolB* also defines two outgoing signals (signal3 and signal4 of types bool and int respectively). On the right side of Figure 138, the structure representation of *ElementA* is shown.

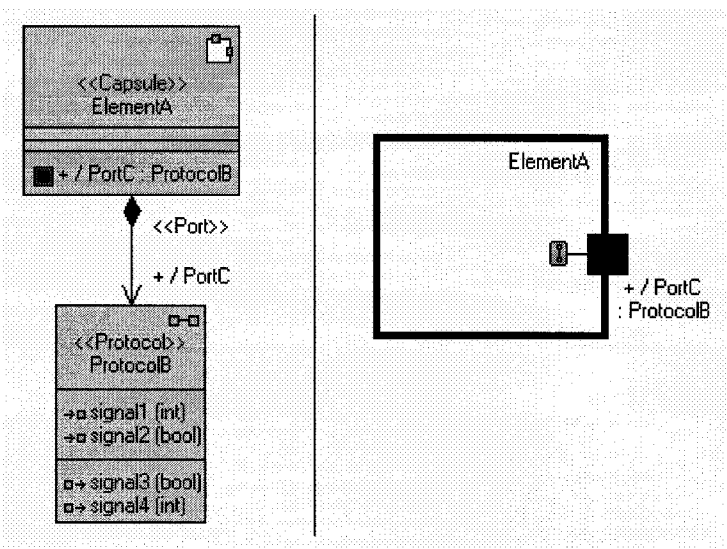


Figure 138. Port definition in ROSE-RT

Figure 139 presents the result of applying the mappings previously defined in this section into the ROSE-RT model of Figure 138. The result is a semantically equivalent representation using UML 2.0 notation. On the left side we see a class diagram with a structured class *ElementA* (mapped from capsule ElementA from Figure 138), which

contains a port *PortC*. The port includes definition of required and provided interfaces *ProtocolB_as_required* and *ProtocolB_as_provided* respectively. The interfaces contain operations mapped upon signals from the protocol class from Figure 138. On the right side of Figure 139, a port representation in UML 2.0 is shown.

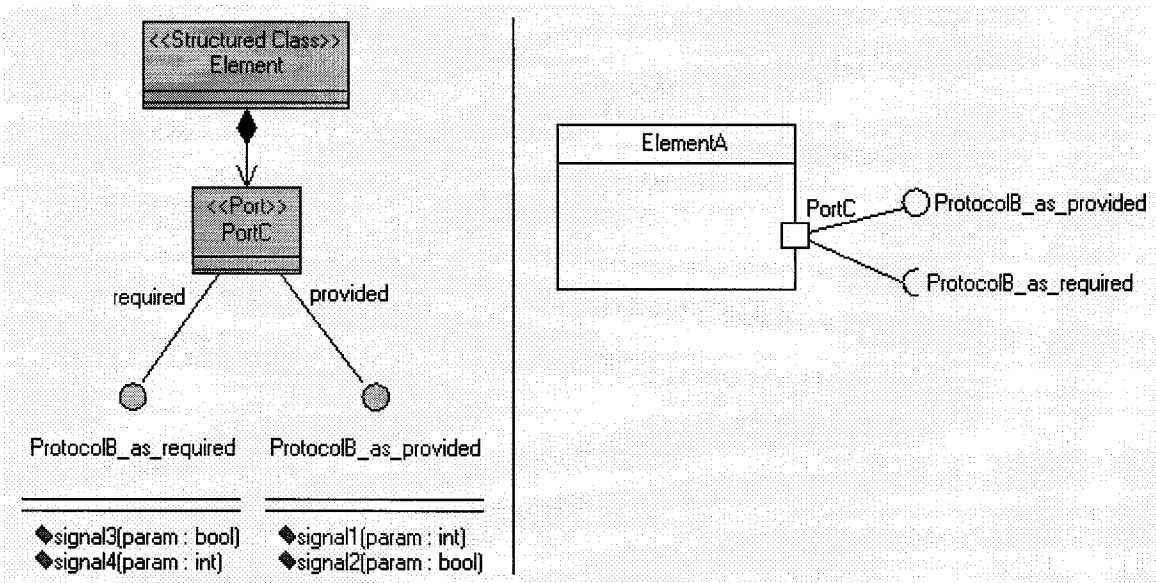


Figure 139. Port Definition in UML 2.0

The ProtocolToInterfaceMapping we just discussed concludes this section. ProtocolToInterfaceMapping is the last mapping that we have consider so far to take our current set of mappings to produce a model using UML 2.0 as a language (that uses structural classes, ports and connectors). We emphasize that there might be some others though. The absence of software tools to read a pure UML 2.0 model (ROSE-RT is very close, but still not pure UML 2.0) prevented us from implementing this new set of mappings and verifying for completeness of a set of UML 1.x to UML 2.0 mapping definitions.

The next chapter presents the conclusion of this thesis.

Chapter 6. Conclusions

In this chapter we first present a summary of this dissertation. Then we offer some conclusions stemming from this work. We conclude with a high level description of future work.

6.1 Summary

In Chapter 1 we first present the context and motivation of this thesis. We work with standard specifications and their eventual compliance verification activities within a model driven development environment. We define our problem as how to verify compliance of a design model against a standard specification model. We give an overview of our proposed solution in which we recognize the fact that there is a semantic gap between specification and design models. We claim that establishing direct links between elements of the specification and design models is complex due to the fact that the process bypasses the origin of aspects that contribute to such gap. We propose to develop a compliance verification approach that addresses such aspects separately. At the core of our proposal lies the creation of a semantically equivalent representation of the specification model, using the same language that is used in the design model. This new model is then used to check the compliance of design models. We first validate our proposal by presenting an extensive case study of the application of our approach to a standard specification of software radios. We also validate our proposal by building prototype software tools to show the feasibility of the approach and strengthen the soundness of our work (through its operationalization). We conclude Chapter 1 by outlining our thesis contributions.

In Chapter 2 we present the background material on which this thesis rests. We start with a brief review of the nature and origin of the gap that may exist between

specification and design models. Among the aspects that contribute to create such a gap we discuss the challenge for compliance verification when different languages are used to define specification and design models. We also discuss the challenge of using different communication paradigms (e.g., procedure-call-based vs. signal-based communications) and nine other aspects that affect model compliance verification. We continue with background information on different international standards that were used in the development of our research. We provide background information on the Object Management Group's Model Driven Architecture (MDA), Unified Modeling Language (UML), Meta Object Facility (MOF), CWM, QVT, OCL and XML). We proceed with background information on Software Radios. We first explain the Software Radio concept, we talk about international efforts to achieve standardization of a software radio specification, and describe the Object Management Group's Software-Based Communication's Software Radio Platform Independent Model. We conclude Chapter 2 with a brief overview of software verification relevant to our work. We first address model to model verification in model driven development, and then give an overview on verifications activities that are being developed in the software radio context.

In Chapter 3 we describe at length our main contributions. We start with our proposed two-step approach for model compliance verification. At the core of the contribution is the introduction of an intermediate model that is semantically equivalent to the specification model but expressed in the same language as the design model. We split transformation and compliance verification activities into two steps. We first address transformation and compliance between the specification model and its semantically equivalent transformed specification (*tSpec*), to subsequently address compliance verification between design and the *tSpec* model. The two steps for compliance verification are composed of several activities. Among the *Spec* to *tSpec* transformation/compliance activities, we define the semantic definition and validation of mappings, the verification of the correct application of mappings and, the verification that the mapping definitions are complete and that every element from the specification model has been taken into consideration as an originator of elements in the *tSpec* model. With respect to compliance verification activities between designs and the *tSpec* model, we

first address structural compliance and subsequently address behavioral compliance. For structural compliance we take into consideration the aspects mentioned in Chapter 2 that contribute to create the gap between specification and design models. Most of our proposed solutions to check structural compliance are based on additional information that is externally provided. For behavioral compliance we rely on Rational Rose RealTime and Rational Quality Architect to attempt to match runtime sequence diagrams that capture design execution against specification level sequence diagrams.

In Chapter 4 we present our case study in software radios. We apply our two-step approach for model compliance verification to a software radio platform independent model (PIM) that we use as our standard specification model. We start defining the source and target metamodels. Even though both the specification and design models are written using UML-based notation, we consider important to specifically identify the type of elements that are used within the specification and design models. We do so to avoid an incomplete set of mappings, and also to prevent allocation of resources towards definitions of mappings that are not to be use. Once the two metamodels have been constructed, we identify semantic relationships between elements of the two metamodels and proceed to write down mapping definitions that will transform a conventional UML 1.5 specification model into a semantically equivalent model expressed using the ROSE-RT notation. We group mappings according to different aspects and provide a brief rationale of the origin of each group. We proceed with an example to verify the correctness of the application of a *Spec* to ROSE-RT mapping. We remind the reader that the same process was applied to each mapping definition from section 4.1.1. We later verify the completeness of our mapping definitions by showing that all elements in the *Spec* model have been taken into consideration as originators of elements in the *tSpec* model. We conclude our case study with two sections where the structural and behavioral compliance activities are performed. Throughout the chapter we describe how our prototype tools help us in our compliance verification efforts.

In Chapter 5 we discuss the generality aspect of our contributions and claim that we have we have reasons to believe that our approach can be generalized. We present

general guidelines for applying our two-step approach for model compliance verification to other standard specifications. We start with a general discussion of the application of our proposed approach to four other OMG standard specifications, and then narrow the scope of the analysis to the use of the Product Lifecycle Management (PLM) standard specification model. We present examples of an attempt to generate a *tSpec* model for the PLM *Spec* model. While analyzing each of the activities of the approach, we highlight the parts that may be reused and point to additional work when required. We conclude the chapter with a section that discusses possible extensions to our mappings if we were to transform a UML 1.4 model into a UML 2.0 that uses structured classes, ports and connectors.

6.2 Conclusion

The complexity of systems currently being developed makes the manual production of new software a more demanding challenge. We believe that MDD presents itself as an excellent candidate to address such a challenge. MDD heavily relies on automation to become a reality, but a type of automation that constitutes a paradigm shift in the way we develop software. We saw a similar shift of paradigm when programming with ones and zeroes was replaced by assembly language. We saw it again when third generation languages emerge as a substitute for assembler programming. We remark that for every such shift of programming paradigm there was the need to create new mechanisms and infrastructure for people to adopt the new programming approach. We believe that our work makes a contribution towards the emergence of MDD. We are proposing an approach to verify model compliance. There is a lot of work ahead as the software engineering community explores this new field.

Our proposal is consistent with MDD inasmuch as relying heavily on automation. In our opinion, our prototype tools do constitute a significant contribution of our work. We partially automated most of our proposed activities. When we needed to identify the *Spec* metamodel, our prototype tools helped us to find the type of elements from the *Spec* model that were not included yet in the *Spec* metamodel. When we needed to verify correctness of our mapping implementations, our tool told us which test cases were not

being tested at all. When we needed to verify that all elements were being used as originators of elements/relationships in the *tSpec* model, again the tool gave us such information. In a sense, we emulated perhaps one of the most successful automation tools in our industry. Compilers are not only meant to catch errors, but also to tell us where they are located in a program, and to suggest why the error occurs.

Transformation of the original *Spec* model was significantly simplified once we started applying the first three activities of our overall approach. We were aiming to produce a semantically equivalent model and the approach allowed us to target specific goals for each iteration. The process guided us to first define a set of mappings, to verify that they were correctly implemented, to define complete test models upon a combinational model and finally to define a complete set of mappings to automatically generate a semantically equivalent *tSpec* model.

With respect to the *tSpec* model itself, its creation proved to be a valuable contribution towards reducing the complexity of verifying structural compliance. Without it, matching the hundreds of elements/relationships from the Design model with those of the *Spec* model would have required to implicitly apply transformations for each element/relationship in the *Spec* model and look for the output of such transformations within the *Design* model. With our semantically equivalent model being automatically generated, transformations are used once and execute separately from structural compliance verification, which can be executed many times without having to transform the specification model again.

Finally, we have shown that standard specification models can be used by automated tools. In other words, we are realizing the MDA vision that demands PIM to PSM transformation using automation of a predefined set of mappings. We have shown that a standard specification in the form of a PIM can be used in such a manner. Still there is plenty of work in the future, and that is what we discuss in the next section of this final chapter.

6.3 Future Work

In this section we discuss future work related to this thesis research. The discussion for each topic is brief as we basically outline the direction or the main idea of each line of work.

6.3.1 QVT defined transformations

As QVT progresses towards becoming a standard in model transformations, refining our work into QVT format seems a natural next step to increase the exposure of our transformation-compliance research in the context of MDA. The consensus that the QVT standard will achieve will increase reusability of our mapping definitions.

The mappings we have defined contain all the elements outlined by QVT to define a mapping. They identify source and target models, they both have preconditions based on OCL constraints, as well as actions to be executed when the mapping is to be applied. It is perhaps the format (i.e., the name of the fields where such information is to be placed) that will vary from our mapping definition to a QVT format.

6.3.2 Off the Shelf model transformation tools

During our work we have developed prototype software tools to help us in the transformation/verification activities. We have proven the feasibility of the approach with our own software tools. The next step would be to consider the possibility of using off-the-shelf transformation tools to implement our mappings, although we may have to wait for them to become available. MDD transformation tools are on the assembly line aiming to provide tool support for QVT. As QVT becomes a standard specification for model transformation, tools supporting QVT will make mapping definitions more portable across different tools and platforms thus increasing the reusability of mapping definitions.

6.3.3 Catalog of UML 1.x to UML 2.0 mappings

With the adoption of the new version 2.0 of the UML specification, there might be significant work related to the migration of software models written on an older version

of UML towards the new 2.0 version of UML. This line of work can significantly reuse our mapping definitions even if they pursue a different purpose. We originally defined mappings to generate a *tSpec* model that would be later used for compliance of a design model. The same procedure can be used to update software models in an older version of UML in order to generate semantically equivalent models in UML 2.0. This would require the definition of new mappings to address UML elements that were not used (and hence for which we did not produce a mapping) in our specification model.

6.3.4 Third party Designs

In our current work, we have used a Design we have developed for our domain (the Software Defined Radio). While this has allowed us to test the whole approach and in particular the tools we have developed, clearly one acid test will be to validate the compliance of a third party *Design* against the transformed specification (*tSpec*) we have obtained from the application of our mappings. As both the third party *Designs* and our *tSpec* model use the same specification model (the SBC's SWRadio PIM) results should be similar as the ones presented in Chapter 4.

6.3.5 Software Radio implementations in different design languages

Ultimately, within the software radio context, compliance testers will not necessarily be given designs conveniently expressed in the same design language as our transformed specification (namely ROSE-RT). Thus, considerable work remains to be done to explore the feasibility of reverse engineering a Software Radio implementation to produce a ROSE-RT Design and verify for compliance.

6.4 In Hindsight

To conclude this dissertation, we briefly discuss an endeavour that extends beyond mere future work and appears to lead to a full-fledged research program we intend to pursue. First, stepping back from what was achieved in this dissertation, we observe that the single most significant drawback of our work is the limited reusability of the mappings that we have developed. More precisely, our mappings are specific to the

modeling languages we consider, that is, to a specification expressed in UML 1.x and a design captured using UML-RT. And we have not dealt with the semantic entirety of either of these modeling languages, but instead limited ourselves mostly to the elements of these languages used for the SCA[15]. The question then is whether this particular contribution can be made more reusable, and, if so, how can this be achieved. There are several facets to this question.

First, let us consider the expression of our mappings. Currently, this involves, in part, the use of C++. It is commonly accepted that such a procedural approach is much less desirable, with respect to reusability, than a declarative one. Consequently, we have recently participated in the supervision of a Master's thesis that follows our approach (*viz.*, from a specification to a *tSpec* to a design) but instead uses QVT [69] (whose semantics are still evolving). That work offers an immediate advantage in terms of mapping definitions. Namely, the use of QVT, *in principle*, provides independence from modeling languages. That is, expressing the mappings (required for a specification in UML 1.x and a design in UML-RT) using QVT not only standardizes the syntax and semantics relevant to such mappings, but also enables the possibility of reusing the same syntax and semantics for mappings between different modeling languages[16].

Comparing our mappings with those obtained by McClean [52] using QVT (within the same scope as our work, namely an SCA-based UML 1.x to UML-RT transformation), leads to a second consideration: the specificity or 'adhocness' of these mappings. Scrutinizing both sets of mappings, we observe that, though they differ in their expression, they *are* conceptually similar. For example, how to map a class onto a capsule works essentially the same way in both approaches. This could be partially due to the fact McClean could take inspiration from our work. But, intuitively, it seems possible, if not likely, that different domain experts would indeed end up with similar mappings. In turn, this suggests that heuristics (pertaining to the creation of mappings) could possibly be 'extracted' from our mappings. Investigating the nature and reusability of such

[15] And, in particular, to structural aspects. Thus, we have ignored UML models such as statecharts.
[16] McClean's thesis [52] mentions this possibility but does not explore it... as this is a significant enterprise.

heuristics is not a trivial question, quite on the contrary! At this point in time, we observe that:

1) Heuristics for the creation of mappings between different modeling languages are semantically vastly different from what are typically called "design heuristics". The latter refer to guidelines promoted by different authors for the creation of 'good' designs. But such guidelines are contextual inasmuch as they depend on promoting certain quality attributes (e.g., modifiability) possibly at the expense of others (e.g., performance). Neither our work, nor the work of McClean pertains to such design heuristics.
2) If mappings between different modeling languages are to be reusable, then they must be separated from specification-specific information. In fact, ideally, reusability of mappings will be maximized if such mappings do not include creator-specific decisions, that is decisions that one mapping creator adopts, but that another would not necessarily reuse. In other words, in hindsight, it would be desirable to somehow distinguish between a) reusable mappings (by virtue of being independent from the idiosyncratic decisions of their creator) and b) specification-specific and creator-specific ones. The separation we make between mappings and so-called "additional information" may provide a simple mechanism to operationalize this distinction.
3) The *a posteriori* extraction of reusable aspects from our work is not a promising avenue. Indeed, McClean's work eloquently demonstrates that a standardized syntax and semantics for the expression of mappings is much more conducive to the separation of reusable from idiosyncratic aspects of mappings than our work can ever be. More precisely, it is because McClean manages to express mappings in terms of QVT notions that he can then investigate which ones are reusable (e.g., across specifications, and across modelling languages) and which ones are not. At this point in time, such investigation has barely started and relies on the conceptual framework of QVT within MDD to determine "how high" a mapping can exist in the hierarchy of models [68] associated with MDD. Our expression of mappings is much more *ad hoc* and, we repeat, therefore less amenable to such a study (and to a comparison to more formal transformational approaches such as correctness-preserving ones).

4) In hindsight, we believe that a "bottom-up" approach that first creates mappings with respect to a certain specification (such as the SCA) and *then* (and only then) investigates which of the mappings may be generalized, may not be as promising as a "top-down" one that would a) make reusability a central issue (which we did not) and b) starting at the top of the MDD hierarchy of modeling languages work down this hierarchy trying to define mappings "as high up" as possible. Let us elaborate. Experience has taught us that it is quite difficult, as creators of mappings, to step back and attempt to generalize. From our viewpoint this task is akin to developing a software system and then asking which parts can be reused. The main difficulty lies in that our focus is on developing *and verifying* a set of *working* mappings. That is, from a development standpoint, it is simpler to have all mappings exist at a same conceptual and operational level than to have the sort of 'hierarchy of mappings' that a top-down approach seems to lead to. This is particularly true with respect to the verification of such mappings: as Binder [6] eloquently demonstrates, hierarchical representations almost systematically need to be flattened in order to be verified. The point to be grasped is that, in the context of MDD, reusability appears to require that mappings be organized hierarchically, but that such an organization not only is hard to obtain *a posteriori* but also may introduce other problems (e.g., with respect to verification).
5) The issue of the reusability of mappings from one modeling language to another, or from one metamodel language to another, or from one metametamodel to another, and so on appears increasingly complex as one studies it. For example, there is a risk that looking for mappings at a very high level of abstraction (such as a metametamodel), becomes a purely semantic exercise[17] deprived of any practicality. Clearly, this is at the opposite of what we wanted to achieve in our work. In other words, we would like to tackle the issue of reusability from a practical viewpoint. In light of the previous discussion, a simple first step consists in having the creators of mappings comment on the reusability of these mappings. Then, MDD reuse experts could decide if a particular mapping (much like a "design pattern") should be

[17] For example, in theory, one should address the necessity and sufficiency of each mapping with respect to the level of the MDD hierarchy to which it belongs, and with respect to the whole hierarchy.

advertised to the rest of the community and also conventionalize the expression of this mapping at the correct level of abstraction.

References

1. Adaptive. *MOF Query / Views / Transformations First Revised Submission.* OMG document ad/03-02-09
2. Alcatel, Softeam, Thales, and TNI-Valiosys. *MOF Query / Views / Transformations First Revised Submission.* OMG document ad/03-08-05
3. S. Bernardi, S. Donatelli and J. Merseguer. *Performance modeling and analysis: From UML sequence diagrams and statecharts to analysable petri net models.* Proceedings of the 3rd International Wworkshop on Software and Performance. 2002
4. J. Bézivin and O. Gerbé. *Towards a precise Definition of the OMG/MDA Framework.* Automated Software Engineering (ASE). 2001.
5. J. Bézivin. *From Object Composition to Model Transformation with the MDA*, Technology of Object-Oriented Languages and Systems (TOOLS), Santa Barbara, California, August 2001.
6. R. V. Binder. *Testing Object-Oriented Systems: Model, Patterns and Tools.* Addison-Wesley 2000.
7. G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide.* Addison-Wesley. 1999
8. F. Bordeleau. *A Systematic and Traceable Progression from Scenario Models to Communicating Hierarchical State Machines.* Ph.D. Thesis. Department of Systems and Computer Engineering. Carleton University. 1999.
9. D. Budgen. *Introduction to software design.* Curriculum module SEI-CM-2.2.1, Carnegie-Mellon University. 1989.
10. R. M. Burstall and J. Darlington. *Some Transformations for Developing Recursive Programs.* Proceedings of International Conference on Reliable Software. IEEE. 1975.
11. R. M. Burstall and J. Darlington. *A Transformation System for Developing Recursive Programs.* Journal of the ACM, Vol. 24, No. 1. 1977.
12. Codagen Technologies Corp. *MOF Query / Views / Transformations First Revised Submission.* OMG document ad/03-03-23
13. Compuware Corporation and SUN Microsystems. *MOF Query / Views / Transformations First Revised Submission.* OMG document ad/03-08-07
14. J. P. Corriveau. *Traceability Process for Large OO Projects.* IEEE Computer. 1996.
15. J. Darlington. *An Experimental Program Transformation and Synthesis System.* Artificial Intelligence Vol. 16. 1981.
16. D. F. DeSouza and A. C. Wills. *Objects, Components, and Frameworks with UML: The Catalysis(SM) Approach.* Addison Wesley Professional. 1999
17. M. Deutsch. *Software Verification and Validation: Realistic Project Approaches.* Prentice Hall. 1982.
18. E. W. Dijkstra. *A Constructive Approach to the Problem of Program Correctness.* BIT 8, No. 3. 1968.
19. E. Domínguez, A. L. Rubio, and M. A. Zapata. *Mapping Models Between Different Modeling Languages.* Proceedings of the ECOOP'2002 Workshop on Integration and Transformation of UML models; J. Araujo, J. Whittle, A. Toval and R. France, Eds. ; Málaga (España), 2002
20. E. Domínguez, M. A. Zapata, and J. J. Rubio. *A Conceptual approach to meta-modeling*, in A. Oliv_e, J.A. Pastor (Eds.), *Adv. Info. Syst. Eng., CAISE'97*, LNCS 1250, Springer,1997
21. E. Domínguez and M. A. Zapata, *Mappings and Interoperability: a Meta-Modeling Approach.* In T. Yakhno (Eds.), Adv. in Info. Syst., ADVIS'00, LNCS 1909, Springer, 2000
22. DSTC, International Business Machines, and CBOP. *MOF Query / Views / Transformations First Revised Submission.* OMG document ad/04-01-06
23. Jean-Marie Favre. Towards a Basic Theory to Model Model Driven Engineering. Workshop in Software Model Engineering (WISME). 2004.

24. D. S. Frankel. *Applying MDA to Enterprise Computing.* OMG Press. 2003.
25. D. S. Frankel. *BPM and MDA: The Rise of Model-Driven Enterprise Systems.* Business Process Trends. Whitepaper. 2003.
26. D. S. Frankel. *Model-Driven Architecture: Reality and Implementation.* OMG MDA Presentation. 2001.
27. P. Freeman and A. I. Wasserman, Editors. *Tutorial on Software Design Techniques, Fourth Edition.* IEEE Catalog No. EHO205-5, IEEE Computer Society Press. 1983.
28. T. Gardner and C. Griffin. *A review of OMG MOF 2.0 Query / Views / Transformations Submissions and Recommendations towards the final Standard.* OMG document ad/03-08-02
29. A. Gerber, M. Lawley, K. Raymond, J. Steel, and A. Wood. *Transformation: The Missing Link of MDA.* 1st International Conference on Graph Transformation. Barcelona (Spain), 2002. LNCS 2505.
30. P. Harmon. *MDA: An Idea Whose Time Has Come.* OMG MDA Presentation. 2002.
31. P.-A. Hsiung, T.-Y. Lee, J.-M. Fu, and W.-B. *Formal Verification of Real-Time Embedded Software in an Object-Oriented Application Framework.* IEE Proceedings Computers and Digital Techniques. Vol. 151, No. 6. 2004.
32. I-Logix Raphsody. www. ilogix.com
33. IBM Rational. www.rational.com.
34. Interactive Objects Software GmbH and Project Technology, Inc. *MOF Query / Views / Transformations First Revised Submission.* OMG document ad/03-08-11
35. International Telecommunication Union. *Information Technology – Open Distributed Processing – Reference Model: Overview.* ITU-T Rec X.901 1997.
36. Institut de Recherce en Informatique et Systèmes Aléatoires (IRISA). http://www.irisa.fr
37. Institut de Recherce en Informatique et Systèmes Aléatoires (IRISA). UMLAUT. http://www.irisa.fr/UMLAUT. 2004.
38. P. Jalote. *An Integrated approach to Software Engineering.* Springer. 1991.
39. Joint Tactical Radio System (JTRS) Joint Program Office (JPO). http://jtrs.army.mil
40. Joint Tactical Radio System Technology Laboratory. https://jtel.spawar.navy.mil/main.asp
41. Joint Tactical Radio System (JTRS) Joint Program Office (JPO). Software Communications Architecture (SCA).
42. Joint Tactical Radio System (JTRS) Mission Description and Budget Item Justification.
43. Kennedy Carter Ltd. *MOF Query / Views / Transformations First Revised Submission.* OMG document ad/03-03-11
44. I. Khriss, R. K. Keller and I. A. Hamid. *Supporting Design by Pattern-based Transformations.* Proceedings of the Second International Workshop on Strategic Knowledge and Concept Formation. 1999.
45. T. Koch, A. Uhl, and D. Weise. *Model Driven Architecture.* Interactive Objects Software GmbH whitepaper. OMG Document ormsc/02-09-04. 2002
46. J. Kovse and T. Härder. *Generic XMI-Based UML Model Transformations.* In: Proceedings OOIS'2002, Montpellier, Sept. 2002, Springer-Verlag
47. S. Leblanc and P. Merle. *Towards Middleware Product-Lines.* Workshop on Model-driven Approaches to Middleware Applications Development (MAMAD) 2003.
48. R. Lemesle. *Transformation Rules Based on Meta-Modeling.* EDOC,'98, La Jolla, California, 1998. http://iae.univ-nantes.fr/recherch/travaux/cahiers98/lemesle.html
49. B. A. Lieberman. *Putting Use Cases to work.* The Rational Edge. February 2002.
50. M. Lindvall and K. Sandahl. *Practical Implications of Traceability.* Software Practice and Experience, vol. 26, no. 10, 1996.
51. A. Mayrhauser. *Software Engineering: methods and management.* Academic Press Professional, Inc. 1990.
52. T. McClean. *Tool Support for Model Compliance Verification: A QVT Based Approach.* Master Thesis. Carleton University. School of Computer Science. January 2005 – To be published.
53. S. J. Mellor and M. J. Balcer. Executable UML: A foundation for Model Driven Architecture. Addison-Wesley. 2002.
54. M. A. de Miguel, D. Exertier, and S. Salicki. *Specification of Model Transformations Based on Meta Templates.* Workshop in Software Model Engineering (WISME). 2002.

55. D. Milicev. *Automatic Model Transformations Using Extended UML Object Diagrams in Modeling Environments.* Technical Report TI-ETF-RTI-00-0042, University of Belgrade, School of Electrical Engineering, October 2000
56. D. Milicev. *Domain Mapping Using Extended UML Object Diagrams.* IEEE Software, Vol. 19, No. 2, March/April 2002
57. E. F. Miller and G. E. Lindamood. *Structured Programming: Top-down Approach.* Datamation. 1973.
58. J. Mitola III. http://ourworld.compuserve.com/homepage/jmitola
59. M. Minea, C. Izbasa, C. Jebelean. *Experience with Formal Verification of SDL Protocols.* Computing International Scientific Journal. Vol. 2, Issue 3. 2003.
60. Modular Software-Programmable Radio Consortium. Support and Rationale Document for the Software Communications Architecture Specification (v2.2).
61. M. Mullerbur, L. Holenderski, O. Maffeis. A. Merceron, and M. Morley. *Systematic Testing and Formal Verification to Validate Reactive Programs.* Software Quality Journal, 4(4), 1995.
62. Objecteering Tests for Java/EJB. www. objecteering.com
63. OMG. *Business Process Management Facility RFP.* Revised Draft RFP. OMG Document bei/02-06-01. 2002.
64. OMG. *Business Process Runtime Interfaces RFP.* OMG Document bei/02-06-08. 2002.
65. OMG. *Common Warehouse Metamodel (CWM) v 1.1 Specification.* OMG Document formal/03-03-02. 2003.
66. OMG. *Data Distribution Service Specification.* Revised Final Adopted Specification. OMG Document ptc/04-04-12. 2004.
67. OMG. *MDA Guide Version 1.0.* Whitepaper. OMG Document omg/03-05-01. 2003.
68. OMG. *Meta Object Facilities (MOF) v.1.4 Specification.* OMG Document formal/02-04-03. 2002.
69. OMG. *MOF 2.0 Query / Views / Transformations RFP.* OMG document ab/02-04-02. 2002.
70. OMG. *A PIM Framework for Telecom OSS.* Proposal to OMG. OMG Document telecom/02-06-06. 2002.
71. OMG. *Product Data Management RFP.* Draft RFP. OMG Document mfg/02-06-02. 2002.
72. OMG. *Product Lifecycle Management Specification.* Draft Adopted Specification. OMG Document dtc/04-05-05. 2004.
73. OMG. *Software Radio Components Specification.* Final Adopted Specification. OMG Document dtc/04-05-04. 2004.
74. OMG. *Super Distributed Object Specification.* Final Adopted Specification. OMG Document dtc/03-09-01. 2003.
75. OMG. *UML 2.0 OCL Specification.* Final adopted specification. OMG Document ptc/03-10-14 2003.
76. OMG. *UML 2.0 Superstructure Specification.* Final adopted specification. OMG Document ptc/03-08-02. 2003
77. OMG Architecture Board. *Model Driven Architecture.* OMG document ormsc/01-07-01. 2001
78. OMG Software-Based Communication (SBC) Domain Task Force (DTF). http://sbc.omg.org
79. OMG Software-Based Communications (SBC) Domain Task Force (DTF). PIM and PSM for SWRadio Components. May 13, 2003.
80. OMG Software-Based Communication (SBC) DomainTask Force (DTF). *SWRadio Platform Independent Model V7 (Rose model).*
81. H. Partsch and R. Steinbruggen. Program Transformation Systems, ACM Computing Surveys, Vol. 15, No. 3. 1983.
82. M. Peltier, J. Bézivin, and G. Guillaume. *MTRANS: A general framework, based on XSLT, for model transformations.* In WTUML'01, Proceedings of the Workshop on Transformations in UML, Genova, Italy, 2001.
83. M. Peltier, F. Ziserman, and J. Bézivin. On levels of model transformation. In *XML Europe 2000,* pages 1–17, Paris, France, June 2000. Graphic Communications Association.
84. D. Pollet, D. Vojtisek, J.-M. Jézéquel. *OCL as a Core UML Transformation Language.* WITUML 2002 Position paper, Malaga, Spain, june, 2002.
85. I. Porres. *A Framework for Model Transformations.* Presented in the Workshop on Integration and Transformation of UML Models - WITUML, Málaga, Spain, June 2002.
86. I. Porres. *A toolkit for manipulating UML models.* Technical Report 441, Turku Centre for Computer Science, 2001. Available at www.tucs.fi.

87. D. Powell. *Formal Methods for Verification Based Software Inspection.* Ph.D. Thesis. Faculty of Engineering and Information Technology. Griffith University. Queensland, Australia. 2002.
88. E. T. Ray. *Learning XML.* O'Reilly 2001.
89. S. Schönberger, R. K. Keller, and I. Khriss. *Algorithmic support for model transformation in object-oriented software development.* Concurrency and Computation: Practice and Experience Vol. 13 No. 5. 2001
90. B. Selic. *The Pragmatics of Model-Driven Development.* IEEE Software. September/October 2003.
91. B. Selic. *Complete High-Performance Code Generation from UML Models.* Embedded Systems Conference. 2002.
92. B. Selic, G. Gullekson, and P. T. Ward. *Real-Time Object-Oriented Modeling*, Wiley Professional Computing.1994.
93. B. Selic and J. Rumbaugh, *Using UML to Model Complex Real- Time Systems*, Rational whitepaper.
94. P. Selonen, K. Koskimies, and M. Sakkinen. *How to Make Apples from Oranges in UML.* In Proc. of HICCS, IEEE Computer Society 2001
95. M. G. Staskauskas . *An experience in the formal verification of industrial software.* Communications of the ACM. Volume 39. Issue 12. 1996.
96. W. P. Stevens, G. J. Myers, and L. L. Constantine. *Structured Design.* IBM Systems Journal, Vol. 13 No. 2, 1974
97. Software Defined Radio Forum (SDRF). http://www.sdrforum.org
98. G. Sunyé, A. Le Guennec, J.-M. Jézéquel. *Using UML Action Semantics for model execution and transformation.* Information Systems. Vol 27. 2002
99. Tata Consultancy Services. *MOF Query / Views / Transformations First Revised Submission.* OMG document ad/03-08-08
100. Telelogic Tau. www.telelogic.com
101. Triskell Team. http://www.irisa.fr/triskell. 2004
102. Triskell Team. *Model Transformation with a Dedicated OO Language.* IRISA Rennes, France 2004.
103. D. Vojtisek and J.-M. Jézéquel. *MTL and Umlaut NG: Engine and Framework for Model Transformation.* ERCIM News. European Research Consortium for Informatics and Mathematics. Number 58, July 2004.
104. Ju An Wang. *Towards Component-Based Software Engineering.* October 2000 Journal of Computing Sciences in Colleges, Volume 16 Issue 1, 2000.
105. A. Warner. *A pragmatical approach to rule-based transformations within UML using XMI.difference.* WITUML: Workshop on Integration and Transformation of UML models. 2002
106. World Wide Web Consortium (W3C). www.w3.org
107. E. Yourdon. Structured Design: Fundamentals of a Discipline of Computer Program and System Design. Prentice-Hall. 1989.

Appendix A. SWRadio Design Compilation Results

```
OutputPath: D:/jpzz/Thesis/AutomatedTransformation/RoseRT/NewComponent1
Running: nmake -nologo RTS_HOME="D:\software\LANGUAGES\Rational\Rose
RealTime/C++/TargetRTS"
MODEL="D:/jpzz/Thesis/AutomatedTransformation/RoseRT/RTModel.rtmdl"
COMPONENT="Component View::NewComponent1" RTmakefiles
        rtcppgen -model "D:/jpzz/Thesis/AutomatedTransformation/RoseRT/RTModel.rtmdl" -component
"Component View::NewComponent1" -makegen
!> Generating Makefiles
!> Generating Makefiles for NewComponent1
Running: nmake -nologo RTcompile
        rtperl "D:\software\LANGUAGES\Rational\Rose RealTime/codegen/chdir_run.pl" src nmake -
nologo -s RTgenerate
!> Generating Package Logical View::RQARTClasses::CPP::V6_4_100_0
!> Generating RQARTAbstractTestBase
!> Generating RQARTAbstractTestHarness
Warning: transition has no events
Warning: transition has no events
!> Generating RQARTAddInProtocol
!> Generating RQARTExceptionCode
!> Generating RQARTIncarnateArguments
!> Generating RQARTStartTestNData
!> Generating RQARTTCPServer
!> Generating Component Unit
!> Generating LifeCycle
!> Generating PortConnector_As_UsesPort_To_Application_As_Disconnector
!> Generating PortSupplier
!> Generating PropertySet
!> Generating Resource
!> Generating ResourceFactory_As_creator_To_Resource_As_product
!> Generating ResourceFactory_As_resourceReleaser_To_Application_As_releaseDelegator
!> Generating ResourceInt
!> Generating Resource_As_assemblyController_To_ApplicationFactory_As_initialConfigurer
!> Generating Resource_As_assemblyController_To_Application_As_delegator
!> Generating Resource_As_deployedResource_To_ApplicationFactory_As_portConnector
!> Generating Resource_As_portUser_To_Resource_As_portProvider
!> Generating Resource_To_Application_As_resourceReleaser
!> Generating runTest_DATA_CLASS_FROM_TestableObject
!> Generating InvalidObjectReference
!> Generating SWRProperties
!> Generating Application
!> Generating ApplicationFactory_As_creator_To_Application_As_product
!> Generating Application_To_ApplicationSequence
!> Generating DeviceAssignmentSequence
!> Generating AdminType_FROM_Device
!> Generating AggregateDevice
!> Generating Device
```

!> Generating DeviceManager
!> Generating DeviceManager_As_informationProvider_To_DomainManager_As_registrar
!> Generating DeviceManager_As_registrar_To_Device_As_registrant
!> Generating DeviceManager_To_DeviceManagerSequence
!> Generating DeviceSequence
!> Generating Device_As_capacityAllocator_To_ApplicationFactory_As_appConfigManager
!> Generating Device_As_capacityDeallocator_To_Application_As_appTeardownManager
!> Generating Device_As_informationProvider_To_DomainManager_As_registar
!> Generating Device_As_product_To_DeviceManager_As_creator
!> Generating Device_To_AggregateDevice
!> Generating Device_To_Device
!> Generating Device_To_DeviceSequence
!> Generating Device_To_Node_Boot_Up
!> Generating Device_To_UsesDevice
!> Generating ExecutableDevice_As_componentTerminator_To_Application_As_appTeardownManager
!> Generating LoadableDevice_As_componentUnloader_To_Application_As_appTeardownManager
!> Generating OperationalType_FROM_Device
!> Generating Comm_User
!> Generating Comm_User_To_Device
!> Generating Node_Boot_Up
!> Generating ServiceSequence
!> Generating Service_As_product_To_DeviceManager_As_creator
!> Generating Service_As_registrant_To_DeviceManager_As_registrar
!> Generating UsageType_FROM_Device
!> Generating ApplicationInstallation
!> Generating ApplicationSequence_FROM_DomainHCI
!> Generating DeviceManagerSequence_FROM_DomainHCI
!> Generating DomainEventChannels
!> Generating DomainHCI
!> Generating DomainManager
!> Generating DomainManager_As_registrar_To_DeviceManager_As_registrant
!> Generating DomainManager_As_registrar_To_Service_As_serviceProvider
!> Generating DomainRegistration
!> Generating
registerDevice_DATA_CLASS_FROM_DomainManager_As_registrar_To_DeviceManager_As_registrant
!> Generating registerDevice_DATA_CLASS_FROM_DomainRegistration
!> Generating registerService_DATA_CLASS_FROM_DomainRegistration
!> Generating registerWithEventChannel_DATA_CLASS_FROM_DomainEventChannels
!> Generating unregisterFromEventChannel_DATA_CLASS_FROM_DomainEventChannels
!> Generating unregisterService_DATA_CLASS_FROM_DomainRegistration
!> Generating DomainManagementObjectAddedEventType
!> Generating DomainManagementObjectEventType
!> Generating DomainManagementObjectRemovedEventType
!> Generating EventChannel_As_eventConsumer_To_Resource_As_eventSupplier
!> Generating EventType
!> Generating
IncomingDomainEventChannel_As_eventConsumer_To_Device_As_eventSupplier_StateChangeEventPro
ducer
!> Generating
IncomingDomainEventChannel_As_eventProducer_To_DomainManager_As_eventConsumer_IncomingD
omainEventConsumer
!> Generating IncomingDomainEventChannel_As_eventsupplier_To_Application_As_eventConsumer
!> Generating
OutgoingDomainEventChannel_As_EventConsumer_To_DomainManager_As_DomainManagementObjec
tRemovedEventProducer

!> Generating
OutgoingDomainEventChannel_As_eventConsumer_To_Application_As_eventSupplier_DomainManage
mentObjectRemovedEventProducer
!> Generating
OutgoingDomainEventChannel_As_eventConsumer_To_DomainManager_As_eventProducer
!> Generating SourceCategoryType
!> Generating FileManager_As_theDomainFileMgr_To_DomainManager_As_creator
!> Generating FileSystem
!> Generating AdministrativeState
!> Generating AvailabilityStatus
!> Generating LogFullAction
!> Generating LogProducer
!> Generating Log_As_logConsumer_To_Application_As_logProducer_LogProducer
!> Generating Log_As_logConsumer_To_DeviceManager_As_logProducer_LogProducer
!> Generating Log_As_logConsumer_To_DomainManager_As_logProducer_LogProducer
!> Generating Log_As_logConsumer_To_Resource_As_logProducer_LogProducer
!> Generating Log_To_DeviceManager_As_logProducer
!> Generating OperationalState
!> Generating ProducerLogRecordSequence
!> Generating EmbeddedNamingService_As_nameRegistrar_To_Application
!> Generating EmbeddedNamingService_As_nameRegistrar_To_DomainManager_As_registrant
!> Generating EmbeddedNamingService_As_nameRegistrar_To_Resource_As_registrant
!> Generating EmbeddedNamingService_To_DeviceManager
!> Generating
DeviceConfigurationDeploymentSpecification_As_specifier_To_DeviceManager_As_specifiedElement
!> Generating
DomainManagerConfigurationSpecification_As_specifier_To_DomainManager_As_specifiedElement
!> Generating EmbeddedCADSpecification_As_specifier_To_Application_As_specifiedElement
!> Generating EmbeddedComponentDeploymentSpecification_To_EmbeddedComponent
!> Generating
ConfigureQueryProperty_As_propertySetConfiguror_To_Resource_As_propertySetConfiguree
!> Generating DeviceArtifactProperty_To_EmbeddedComponent
!> Generating ExecutableProperty_As_executableArguments_To_EmbeddedComponent_As_mainProcess
!> Generating TestProperty_As_configuror_To_Resource_As_configuree
!> Generating RQARTTestHarnessTestRun
!> Generating EmbeddedComponent
!> Generating EmbeddedComponent_As_deployedElement_To_Application_As_container
!> Generating EmbeddedComponent_As_product_To_ApplicationFactory_As_creator
!> Generating ResourcePort_To_Component
!> Generating ComponentElementSequence
!> Generating ComponentProcessIdSequence
!> Generating Component
!> Generating Component_To_Component
!> Generating Component_To_DeploymentSpecification
!> Generating SWRString
!> Generating Property_To_Component
rtperl "D:\software\LANGUAGES\Rational\Rose RealTime/codegen/chdir_run.pl" build nmake -
nologo -k -s RTcompile
!> Compiling Application
Application.cpp
../src/Application.cpp(247) : error C4716: 'Application_Actor::getProfile' : must return a value
../src/Application.cpp(255) : error C4716: 'Application_Actor::getName' : must return a value
../src/Application.cpp(263) : error C4716: 'Application_Actor::getComponentNamingContexts' :
must return a value
../src/Application.cpp(271) : error C4716: 'Application_Actor::getComponentProcessIds' : must
return a value

../src/Application.cpp(279) : error C4716: 'Application_Actor::getComponentDevices' : must return a value
../src/Application.cpp(287) : error C4716: 'Application_Actor::getComponentImplementations' : must return a value
NMAKE : warning U4010: 'Application.OBJ' : build failed; /K specified, continuing ...
!> Compiling AggregateDevice
AggregateDevice.cpp
../src/AggregateDevice.cpp(235) : error C4716: 'AggregateDevice_Actor::getDevices' : must return a value
NMAKE : warning U4010: 'AggregateDevice.OBJ' : build failed; /K specified, continuing ...
!> Compiling Device
Device.cpp
../src/Device.cpp(247) : error C4716: 'Device_Actor::allocateCapacity' : must return a value
../src/Device.cpp(263) : error C4716: 'Device_Actor::getUsageState' : must return a value
../src/Device.cpp(271) : error C4716: 'Device_Actor::getOperationalState' : must return a value
../src/Device.cpp(287) : error C4716: 'Device_Actor::getAdminState' : must return a value
../src/Device.cpp(295) : error C4716: 'Device_Actor::getSoftwareProfile' : must return a value
../src/Device.cpp(303) : error C4716: 'Device_Actor::getLabel' : must return a value
../src/Device.cpp(311) : error C4716: 'Device_Actor::getCompositeDevice' : must return a value
NMAKE : warning U4010: 'Device.OBJ' : build failed; /K specified, continuing ...
!> Compiling DeviceManager
DeviceManager.cpp
../src/DeviceManager.cpp(320) : error C4716: 'DeviceManager_Actor::getComponentImplementationId' : must return a value
../src/DeviceManager.cpp(328) : error C4716: 'DeviceManager_Actor::getDeviceConfigurationProfile' : must return a value
../src/DeviceManager.cpp(336) : error C4716: 'DeviceManager_Actor::getFileSystem' : must return a value
../src/DeviceManager.cpp(344) : error C4716: 'DeviceManager_Actor::getIdentifier' : must return a value
../src/DeviceManager.cpp(352) : error C4716: 'DeviceManager_Actor::getLabel' : must return a value
../src/DeviceManager.cpp(360) : error C4716: 'DeviceManager_Actor::getRegisteredDevices' : must return a value
../src/DeviceManager.cpp(368) : error C4716: 'DeviceManager_Actor::getRegisteredServices' : must return a value
NMAKE : warning U4010: 'DeviceManager.OBJ' : build failed; /K specified, continuing ...
!> Compiling Device_As_product_To_DeviceManager_As_creator
Device_As_product_To_DeviceManager_As_creator.cpp
!> Compiling Device_To_AggregateDevice
Device_To_AggregateDevice.cpp
!> Compiling Device_To_Device
Device_To_Device.cpp
!> Compiling Comm_User
Comm_User.cpp
!> Compiling Comm_User_To_Device
Comm_User_To_Device.cpp
!> Compiling DomainManager
DomainManager.cpp
!> Compiling DomainManager_As_registrar_To_DeviceManager_As_registrant
DomainManager_As_registrar_To_DeviceManager_As_registrant.cpp
!> Compiling DomainRegistration
DomainRegistration.cpp
!> Compiling
registerDevice_DATA_CLASS_FROM_DomainManager_As_registrar_To_DeviceManager_As_registrant
registerDevice_DATA_CLASS_FROM_DomainManager_As_registrar_To_DeviceManager_As_registrant
.cpp

```
!> Compiling registerDevice_DATA_CLASS_FROM_DomainRegistration
registerDevice_DATA_CLASS_FROM_DomainRegistration.cpp
!> Compiling registerService_DATA_CLASS_FROM_DomainRegistration
registerService_DATA_CLASS_FROM_DomainRegistration.cpp
!> Compiling RQARTTestHarnessTestRun
RQARTTestHarnessTestRun.cpp
!> Compiling NewComponent1
NewComponent1.cpp
NMAKE : warning U4011: 'RQARTTestHarnessTestRun.EXE' : not all dependents available; target not
built
NMAKE : warning U4011: 'RTcompile' : not all dependents available; target not built
NMAKE : fatal error U1077: 'rtperl' : return code '0x1'
Stop.
Error: Build failed status 2

Build failed.
```