Mälardalen University Press Doctoral Thesis
No.148

From Models to Code and Back: A Round-trip Approach for Model-driven Engineering of Embedded Systems

Federico Ciccozzi

January 2014

School of Innovation, Design and Engineering
Mälardalen University
Västerås, Sweden

Copyright © Federico Ciccozzi, 2014
ISSN 1651-4238
ISBN 978-91-7485-129-8
Printed by Mälardalen University, Västerås, Sweden

Abstract

The complexity of modern systems is continuously growing, thus demanding novel powerful development approaches. In this direction, model-driven and component-based software engineering have reached the status of promising paradigms for the development of complex systems. Moreover, in the embedded domain, their combination is believed to be helpful in handling the ever-increasing complexity of such systems. However, in order for them and their combination to definitively break through at industrial level, code generated from models through model transformations should preserve system properties modelled at design level.

This research work focuses on aiding the preservation of system properties throughout the entire development process across different abstraction levels. Towards this goal, we provide the possibility of analysing and preserving system properties through a development chain constituted of three steps: (i) generation of code from system models, (ii) execution and analysis of generated code, and (iii) back-propagation of analysis results to system models. With the introduction of steps (ii) and (iii), properties that are hard to predict at modelling level are compared with runtime values and this consequently allows the developer to work exclusively at modelling level thus focusing on optimising system models with the help of those values.

Sammanfattning

Denna doktorsavhandling presenterar nya och förbättrade tekniker för modelldriven och komponentbaserad utveckling av programvara. Syftet är att bevara systemegenskaper, som specificerats i modeller, genom de olika stadierna av utvecklingen och när modeller översätts mellan olika abstraktionsnivåer och till kod. Vi introducerar möjligheter att studera och bevara systemets egenskaper genom att skapa en kedja i tre steg som: (i) genererar kod från systemmodellen, (ii) exekverar och analyserar den genererade koden och (iii) slutligen återkopplar analysvärden till systemmodellen. Introduktionen av steg (ii) och (iii) gör det möjligt att genomföra en detaljerad analys av egenskaper som är svåra, eller till och med omöjliga, att studera med hjälp av endast systemmodeller.

Fördelen med det här tillvägagångssättet är att det förenklar för utvecklaren som slipper arbeta direkt med kod för att ändra systemegenskaper. Istället kan utvecklaren arbeta helt och hållet med modeller och fokusera på optimering av systemmodeller med hjälp av analysvärden från testkörningar av systemet. Vi är övertygade om att denna typ av teknik är nödvändig att utveckla för att stödja modelldriven utveckling av programvara eftersom dagens tekniker inte möjliggör för systemutvecklare att specificera, analysera och optimera systemegenskaper på modellnivå.

Prefazione

La continua crescita in complessitá dei sistemi software moderni porta alla necessitá di definire nuovi e piú efficaci approcci di sviluppo. In questa direzione, metodi basati su modelli (model-driven engineering) e componenti (component-based software engineering) sono stati riconosciuti come promettenti nuove alternative per lo sviluppo di sistemi complessi. Inoltre l'interazione tra loro é ritenuta particolarmente vantaggiosa nella gestione nello sviluppo di sistemi integrati. Affinché questi approcci, cosí come la loro interazione, possano definitivamente prendere piede in campo industriale, il codice generato dai modelli tramite apposite transformazioni deve essere in grado di preservare le proprietá di sistema, sia funzionali che extra-funzionali, definite nei modelli.

Il lavoro di ricerca presentato in questa tesi di dottorato si focalizza sul preservamento delle proprietá di sistema nell'intero processo di sviluppo e attraverso i diversi livelli di astrazione. Il risultato principale é rappresentato da un approccio automatico di round-trip engineering in grado di sostenere il preservamento delle proprietá di sistema attraverso: 1) generazione automatica di codice, 2) monitoraggio e analisi dell'esecuzione del codice generate su piattaforme specifiche, e 3) offrendo la possibilitá di propagare verticalmente i risultati da runtime al livello di modellazione. In questo modo, quelle proprietá che possono essere stimate staticamente solo in maniera approssimativa, vengono valutate in rapporto ai valori ottenuti a runtime. Ció permette di ottimizzare il sistema a livello di design attraverso i modelli, piuttosto che manualmente a livello di codice, per assicurare il preservamento degli proprietá di sistema d'interesse.

Acknowledgements

There are many people to thank for making the path towards this doctoral thesis possible and pleasant. I would like to start with my family that always believed in me and supported, even economically, my decision to move to Sweden for pursuing my objectives; without them I would not have been able to follow my instinct and achieve this result.

A special thanks goes to my main supervisor Mikael Sjödin and assistant supervisor Antonio Cicchetti that have continuously supported me, my curiosity and talkativeness, helping and driving me for achieving very satisfactory results. I would like to thank all my colleagues at MDH, especially at IDT, for all the moments, both fun and constructive, we spent together hoping that many more are to come in the near future.

Being able to stand me and my vim, I would like to thank my office room mate Mehrdad. Many friends, both in Italy and Sweden, have believed in me and made it possible for me to fully enjoy every moment in my free time spent with them; a special thanks goes to all of them. Last but not least, thanks for standing by my side, believing in me and sharing her everyday life with me goes to my girlfriend Julia.

Federico Ciccozzi
Västerås, December, 2013

List of Publications

Main Contributing Publications

Automatic Synthesis of Heterogeneous CPU-GPU Embedded Applications from a UML Profile, Federico Ciccozzi, 6th International Workshop on Model Based Architecting and Construction of Embedded Systems (ACES-MB) at MODELS, Miami, USA, September, 2013.

Towards Code Generation from Design Models for Embedded Systems on Heterogeneous CPU-GPU Platforms, Federico Ciccozzi, IEEE International Conference on Emerging Technology and Factory Automation (ETFA) – Work in Progress Session, IEEE, Cagliari, Italy, September, 2013.

Towards Translational Execution of Action Language for Foundational UML, Federico Ciccozzi, Antonio Cicchetti, Mikael Sjödin, Proceedings of the 39th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), IEEE, Santander, Spain, September, 2013.

An Automated Round-trip Support Towards Deployment Optimization in Component-based Embedded Systems, Federico Ciccozzi, Mehrdad Saadatmand, Antonio Cicchetti, Mikael Sjödin, Proceedings of the 16th International ACM SIGSOFT Symposium on Component Based Software Engineering (CBSE), Vancouver, Canada, June, 2013.

From Models to Code and Back: Correct-by-construction Code from UML and ALF, Federico Ciccozzi, ACM Student Research Competition (SRC) at the International Conference of Software Engineering (ICSE), ACM, San Francisco, USA, May, 2013.

Exploiting UML Semantic Variation Points to Generate Explicit Component Interconnections in Complex Systems, Federico Ciccozzi, Antonio Cicchetti, Mikael Sjödin, Proceedings of the International Conference on Information Technology: New Generations (ITNG), IEEE, Las Vegas, USA, April, 2013.

Full Code Generation from UML Models for Complex Embedded Systems, Federico Ciccozzi, Antonio Cicchetti, Mikael Sjödin, Second International Software Technology Exchange Workshop (STEW) 2012, Swedsoft, Kista, Stockholm (Sweden), November, 2012.

Round-Trip Support for Extra-functional Property Management in Model-Driven Engineering of Embedded Systems, Federico Ciccozzi, Antonio Cicchetti, Mikael Sjödin, Journal of Information and Software Technology (INFSOF), Elsevier, 2012.

Enhancing the Generation of Correct-by-construction Code from Design Models for Complex Embedded Systems, Federico Ciccozzi, Mikael Sjödin, IEEE International Conference on Emerging Technology and Factory Automation (ETFA) - Work in Progress Session, IEEE, Krakow, Poland, July, 2012

Generation of Correct-by-Construction Code from Design Models for Embedded Systems, Federico Ciccozzi, Antonio Cicchetti, Mikael Krekola, Mikael Sjödin, Work-In-Progress at IEEE International Symposium on Industrial Embedded Systems (SIES), Västerås, Sweden, 2011.

Toward a Round-Trip Support for Model-Driven Engineering of Embedded Systems, Federico Ciccozzi, Antonio Cicchetti, Mikael Sjödin, EUROMICRO Conference on Software Engineering & Advanced Applications (SEAA), Oulu, Finland, August, 2011. **Best Paper Award**.

Evolution Management of Extra-Functional Properties in Component Based Embedded Systems, Antonio Cicchetti, Federico Ciccozzi, Thomas Leveque, Séverine Sentilles, International ACM SIGSOFT Symposium on Component Based Software Engineering (CBSE), Boulder, Colorado (USA), June, 2011.

Related Publications

Towards a Novel Model Versioning Approach based on the Separation between Linguistic and Ontological Aspects, Antonio Cicchetti, Federico Ciccozzi, International Workshop on Models and Evolution (ME) at MODELS, Miami, USA, September, 2013.

Towards Migration-Aware Filtering in Model Differences Application, Federico Ciccozzi, Antonio Cicchetti, International Workshop on Models and Evolution (ME) at MODELS, Innsbruck, Austria, October, 2012.

A hybrid approach for multi-view modeling, Antonio Cicchetti, Federico Ciccozzi, Thomas Leveque, Journal of Electronic Communications of the EASST, EASST, June, 2012.

A Solution for Concurrent Versioning of Metamodels and Models, Antonio Cicchetti, Federico Ciccozzi, Thomas Leveque, Journal of Object Technology (JOT), AITO, August, 2012.

CHESS: a Model-Driven Engineering Tool Environment for Aiding the Development of Complex Industrial Systems, Antonio Cicchetti, Federico Ciccozzi, Silvia Mazzini (Intecs SpA), Stefano Puri (Intecs SpA), Marco Panunzio (University of Padova), Tullio Vardanega (University of Padova), Alessandro Zovi (University of Padova), 27th International Conference on Automated Software Engineering (ASE), Essen, Germany, September, 2012.

Supporting Incremental Synchronization in Hybrid Multi-View Modeling, Antonio Cicchetti, Federico Ciccozzi, Thomas Leveque, ACM/IEEE International Conference on Model Driven Engineering Languages & Systems (MODELS), Wellington, New Zealand, October, 2011. **Best Paper Award**.

A Hybrid Approach for Multi-View Modeling, Antonio Cicchetti, Federico Ciccozzi, Thomas Leveque, International Workshop on Multi-Paradigm Modeling (MPM) at MODELS, Wellington, New Zealand, October, 2011.

On the concurrent Versioning of Metamodels and Models: Challenges and possible Solutions, Antonio Cicchetti, Federico Ciccozzi, Thomas Leveque, Alfonso Pierantonio, International Workshop on Model Comparison in Practice (IWCMP), Zurich, Switzerland, June, 2011.

An Open-Source Pivot Language for Proprietary Tools Chaining, Antonio Cicchetti, Federico Ciccozzi, Stefano Cucchiella, International Workshop on Model-Based Development for Computer-Based Systems - Covering Domain and Design Knowledge in Models (ECBS-MBD), Las Vegas, Nevada, USA, April, 2011.

CHESS Tool presentation, Antonio Cicchetti, Federico Ciccozzi, Mikael Krekola, Silvia Mazzini, Marco Panunzio, Stefano Puri, Carlo Santamaria, Tullio Vardanega, Alessandro Zovi, TOPCASED Days, Toulouse, France, February, 2011.

Automating Test Cases Generation: From xtUML System Models to QML Test Models, Federico Ciccozzi, Antonio Cicchetti, Toni Siljamäki, Jenis Kavadiya, Workshop on Model-based Methodologies for Pervasive and Embedded Software (MOMPES) at ASE, Antwerp, Belgium, September, 2010.

Other Publications

Multi-dimensional Assessment of Risks in a Distributed Software Development Course, Ivana Bosnic (University of Zagreb), Federico Ciccozzi, Igor Cavrak (University of Zagreb), Marin Orlic (FER, University Zagreb, Croatia), Raffaela Mirandola (Politecnico di Milano), CTGDSD Workshop at the International Conference on Software Engineering (ICSE), ACM, San Francisco, USA, May, 2013.

Integrating Wireless Systems into Process Industry and Business Management, Federico Ciccozzi, Antonio Cicchetti, Tiberiu Seceleanu, Johan Åkerberg, Lars Eric Carlsson, Jerker Delsing, International Conference on Emerging Technology and Factory Automation (ETFA), Bilbao, Spain, September, 2010.

Performing a project in a Distributed Software Development Course: Lessons Learned, Federico Ciccozzi, Ivica Crnkovic, International Conference on Global Software Engineering (ICGSE), Princeton, New Jersey, USA, August, 2010.

To my family

The problem in this business isn't to keep people from stealing your ideas; it's making them steal your ideas!

Howard H. Aiken

Contents

Chapter 1

Introduction

The intricacy of complex embedded systems demands proper development mechanisms able to effectively deal with it. Towards this purpose, Model-Driven Engineering (MDE) [1] and Component-Based Software Engineering (CBSE) [2] have earned consideration for their ability to mitigate software-development complexity by tackling different issues with dedicated solutions. More specifically, the former shifts the focus of the development from hand-written code to models from which the implementation is meant to be automatically generated through the exploitation of model transformations. The latter breaks down the set of desired features and their intricacy into smaller replaceable sub-modules, namely components, starting from which the application can be built-up and incrementally enhanced. Moreover, their combination has been recognised as an enabler for them to definitely break through for industrial development of embedded systems [3].

Among the others, one of the core goals of MDE is the provision of automated code generation from design models; however, this goal is too often seen as the very final step of an MDE approach [4]. On the one hand, preservation of extra-functional properties (EFPs) throughout the development process by means of appropriate description and verification is of crucial importance since it allows to reduce final product verification and validation effort and costs by providing *correctness-by-construction*, which opposes the more costly *correctness-by-correction* typical of code-centric approaches. On the other hand, certain EFPs specified at modelling level are difficult to determine without code generation and execution [5]. That is the reason for which these properties need to be measured at code level through monitoring or analysis

activities [6]. This would be the case, e.g., of performance-related EFPs, that often only emerge in a running product. As an example, let us consider two sorting algorithms that speed up a program because they use a big portion of the main memory. Although both increase the performance in isolation and they have no direct functional interaction, in combination they may degrade the overall performance because both share the same (too small) main memory [7].

The outcome of this research work is a novel model-driven technique that aids the preservation of system properties from models to generated code. On the one side, one could argue that MDE is highly suitable for the management of system properties thanks to the promotion of their modelling and early analysis. On the other side, very little has been achieved, or even attempted, in practice when it comes to ensuring preservation of system properties (especially EFPs) when transforming models for, e.g., code generation purposes. To the best of our knowledge, no work has previously introduced the notion of back-propagation across different abstraction levels (i.e., from runtime to model) to evaluate preservation of EFPs from models to generated code. The proposed technique is represented by a round-trip approach which consists of the combination of the following four steps:

- Modelling: the first step is represented by modelling the system through a structural design in terms of components, a behavioural description by means of state-machines and action code, as well as a deployment model describing the allocation of software components to operating system's processes;
- Code generation: from the information contained in the design model, we automatically generate full functional code. Note that we refer to generated code as *full* or *full-fledged* if it is entirely generated in an automated manner and does not require manual tuning in order to be executed on the selected platform;
- Monitoring: after the code has been generated we monitor its execution on the target platform and measure selected EFPs;
- Back-propagation: at this point, gathered values are back-propagated to the modelling level and, after their evaluation, the design model can be manually tuned to generate, e.g., more resource-efficient code.

Moreover, we show how the approach can be employed in order to use the measurements gathered at system implementation (or runtime) level for deployment assessment at modelling level. Also in this case, no previous attempt

has been found in the literature that employs measurements gathered at system implementation level to assist the developer in taking deployment decisions at modelling level.

1.1 Basic Concepts

In this section we introduce the basic concepts upon which we build our research.

Model-Driven Engineering and Component-Based Software Engineering. The core concept in Model-Driven Engineering (MDE) is the *model*, considered as an abstraction of the system under development. Rules and constraints for building models have to be properly described through a corresponding language definition and in this respect, a *metamodel* describes the set of available concepts and well-formedness rules a correct model must conform to [8].

Following the MDE paradigm, a system is developed by designing models and refining them starting from higher and moving to lower levels of abstraction until code is generated; refinements are performed through transformations between models. A *model transformation* translates a source model to a target model while preserving their well-formedness [9]. More specifically, in this research work we exploit the following kinds of model transformation:

- Model-to-model (M2M): translates between source and target models, which can be instances of the same or different languages, often exploiting syntactic typing of variables and patterns. Different approaches exist: direct manipulation, relational, graph-based, structure-driven and hybrid. In this research work we exploit mainly hybrid approaches combining direct and relational manipulations;

- Model-to-text (M2T): a particular case of M2M where the target artefact is represented by text. Two main approaches exist: template-based, where a template represents the target text with holes for variable parts computed at runtime with metacode, and visitor-based, where simple visitor mechanisms are defined to traverse the internal representation of a model and write text to a text stream. In our solution we exploit template-based mechanisms for the generation of code from models;

- Text-to-model (T2M): in this case the transformation operates in the opposite direction as the M2T, generating a model from a textual represen-

tation. In this research work we mainly utilize T2M transformations for in-place modifications.

Any of these types of model transformation may be defined as *in-place*, meaning that source (or one of the sources) and target are represented by the same model; in this case, the transformation provides as output an updated version of (one of) the model(s) in input. Most of the transformations described in this thesis, except for the in-place transformations which are by nature *endogenous*, are *exogenous* meaning that they operate between artefacts expressed using different languages [9].

As base for our work we employ the increasingly popular synergy of MDE and Component-Based Software Engineering (CBSE). Since different nuances of the CBSE-related terminology can be found in the literature, in this work we exploit the *component-based design* pattern as prescribed by the UML Superstructure [10], leaving other specific aspects related to CBSE as future direction (as explained in Section 11.2). That is to say, a system is modelled as an assembly of components communicating via required and provided interfaces exposed by ports, where a port represents an interaction between a classifier instance and its internal or external environment. Additionally, features owned by required interfaces are meant to be offered by one or more instances of the owning classifier to one or more instances of the classifiers in its internal or external environment.

Extra-functional Properties, Monitoring and Preservation. In this work, as well as in the related publications, we employ the term *Extra-functional Property* (EFP) as synonym for Non-functional Property. Hence, for EFP we intend those properties that define the overall quality attributes of the system. Moreover, EFPs can set restrictions on the product being developed, as well as on the development process itself, by specifying constraints that must be met. Examples of EFPs include safety, security, usability, reliability and performance.

A fairly wide assortment of different approaches devoted to the measurement of EFPs at system implementation level exists. In this work we focus on *runtime monitoring*, that represents a method to observe the execution of a system in order to determine whether its actual behaviour is in compliance with the intended one. In comparison to other verification techniques such as static analysis, model checking, testing and theorem proving which are used mainly to determine "universal correctness" of software systems, runtime monitoring focuses on each instance and current execution of a system [11].

Preservation of system properties entails the ability to ensure that what is defined at modelling level both functionally and extra-functionally is actually reflected in the generated implementation. To achieve that, the research described in this thesis focused on providing a complete model-driven mechanism for code generation and back-propagation of monitoring results from code execution. In this work we consider the following performance-related EFPs: execution time, response time, heap and stack memory usage. The choice of EFPs was driven by the monitoring possibilities provided by the target platforms as well as the modelling concepts able to host the back-propagated values.

Correctness-by-construction. For correctness-by-construction we refer to the ability to demonstrate or argue software correctness in terms of the approach exploited to generate it. A correct-by-construction approach means that the requirements are *more likely* to be met, the system is *more likely* to be the correct system to meet the requirements, the implementation is *more likely* to be defect-free, and upgrades are *more likely* to retain the original correctness properties. In this respect it is worth noting that the notion of *correctness* refers, in this research work, to the adherence of the generated code to what was specified at model level, once the generation process (i.e., model transformations) has been validated [12]. Nonetheless the correctness of the user solution in the modelling space must be demonstrated for every model, for instance by adopting model verification methods. Possible model-based analysis techniques can be employed for this purpose, even though the verification of the models goes beyond the scope of our contribution.

CHESS Modelling Language. As reference modelling language in this work we employ the cross-domain Composition with Guarantees for High-integrity Embedded Software Components Assembly (CHESS) [13] modelling language (CHESS-ML) [14]. The CHESS-ML has been defined as a UML [15] profile, including tailored subsets of the SysML [16] profile, for requirements definition, and the MARTE [17] profile for extra-functional as well as deployment modelling. The CHESS tool environment has been developed as a set of Eclipse plugins on top of MDT Papyrus [18], an open source integrated environment for editing EMF [19] models and particularly supporting UML and related profiles such as SysML and MARTE, on the Eclipse platform.

CHESS-ML allows the specification of a system together with some EFPs such as predictability, dependability, and security. Moreover, it supports a development methodology expressly based on separation of concerns; distinct

design views address distinct concerns. In addition, CHESS actively supports component-based development. The CHESS component model is conceived in a manner that permits domain-specific needs to be addressed by adding specialization features to a domain-neutral core. In this manner CHESS intends to support a variety of application domains, the common character of which is to embrace model-driven engineering solutions for the development of dependable and predictable real-time embedded systems. According to the CHESS methodology, functional and extra-functional characteristics of the system are defined in specific separated views as follows:

- Functional: the development style follows the component-based pattern where each component is equipped with provided and required interfaces realised via ports and with state-machines and other standard UML diagrams to express functional behaviour. Regarding state-machines, which we employ in this research work for modelling behavioural aspects, neither hierarchical nested states nor orthogonal regions [10] are considered. Moreover, the OMG Action Language for Foundational UML (ALF) [20] is used to enrich the behavioural description. In this way, we reach the necessary expressive power to be able to generate 100% of the implementation directly from the models;

- Extra-functional: in compliance with the principle of separation of concerns adopted in CHESS, the functional models are decorated with extra-functional information thereby ensuring that the definition of the functional entities is not altered.

Moreover, the set of design views is completed by: *Requirement View*, used to model the software requirements and associate them to other model entities, *Deployment View*, which supports the modelling of the target execution platform and software to hardware components allocations, *Analysis View*, that is a set of subviews in which the user can model the analysis contexts used as input for the analysis tools. The latter is split in two distinct views, each specialised for a given type of analysis: (i) dependability (*Dependability Analysis* view) and (ii) predictability (*RT Analysis* view). Most importantly, for each technique back-propagation features have been implemented for enriching the design models with the analysis results, thus enabling a multi-perspective extra-functional evaluation of the system.

The distinct design views are automatically generated by the tool as UML packages in the Papyrus Model Explorer when the user creates a new CHESS model. Switching from a design view to another causes the change of a view

indicator (set of different colours each representing a different view) placed on the main toolbar. The user can even switch from functional to extra-functional view and thereby model extra-functional definitions by means of decorations of already modelled functional entities. Switching among views affects also the model entities available in a customised CHESS palette; this enforces separation of concerns by driving the user in choosing among a set of entities (changing from view to view) in each of the different design phases.

Concerning model-based analysis, state-based, Failure Propagation Transformation Calculus (FPTC), Failure Mode Effects & Criticality (FMECA), Failure Mode and Effect (FMEA) and Fault Tree (FTA) are the means through which CHESS supports different kinds of evaluation of the dependability attributes of the system [21, 22]. Moreover, schedulability analysis is provided to verify whether the timing requirements set on interfaces can be met [23]. The extraction of information from the user model (i.e., generation of a Platform-Specific Model, PSM, or a Schedulability Analysis Model, SAM) and generation of the input for the analysis tools are automated; the results of the analysis are propagated back to the design model as read-only attributes of the appropriate design entities. Thanks to the full automation, as well as the code generation and monitoring mechanisms described in this work, the analysis can be iterated at will until the designer is satisfied with the result.

OSE Real-Time Operating System. OSE is a commercial and industrial real-time operating system developed by Enea[1] which has been designed from the ground specifically for fault-tolerant and distributed systems. It is widely adopted mainly in telecommunication domain for systems ranging from mobile phones to radio base stations [24]. OSE provides the concept of direct and asynchronous message passing for communication and synchronisation between tasks, and its programming model is based on this concept. This allows tasks to run on different processors or cores, utilising the same message-based communication model as on a single processor. This programming model provides the advantage of avoiding the use of shared memory among tasks. In OSE, the runnable real-time entity equivalent to a task is called *process*, and the messages that are passed between processes are referred to as *signals* (thus, the terms *process* and *task* in this thesis can be considered synonyms).

[1] `http://www.enea.com`

1.2 Research Goal and Challenges

In order for code generators, and generally MDE, to be adopted in industrial settings, preservation of system properties throughout the development process by means of appropriate description and verification is pivotal. The way towards this adoption is often undermined by the clash of the common misconception that code generation always represents the very final step of an MDE approach and the fact that, in the embedded domain, certain EFPs are extremely hard or even impossible to be accurately predicted at modelling level without code execution. As a solution to this clash, this research work provides *an automated round-trip approach for the preservation of system properties throughout the development of embedded systems*.

For achieving this goal, the following research challenges have been formulated and considered as main drivers for the work presented in this thesis:

Research Challenge 1 (RC1) – Define an automated process to enable the generation of full-fledged code from design models
Automating the generation of full code concerns the manipulation of design models to generate target code through transformation mechanisms. Moreover, exploiting UML profiles leads to the need of handling the UML's fuzziness, in terms of undecided semantics, in a proper manner. For instance, in the case of component-based design pattern in UML, while the number of instances of components and ports can be precisely specified, the port-to-port links are not equipped with a detailed specification of the component instances they connect.

In order to provide full code generation we need to be able to automatically generate the set of links between explicit component instances and therefore the solution cannot prescind from adding the semantic information needed to drive the definition of links' source and target. In this sense, we need to: (i) define semantic rules for driving the generation of links between explicit component instances via ports, (ii) identify appropriate means for storing the generated information, (iii) and perform the actual generation following the defined rules as well as the hierarchical composition of components and ports.

Research Challenge 2 (RC2) – Define and implement translational execution of ALF towards non-UML platforms
In our approach, complex behavioural descriptions are defined through ALF action code in the models; this information needs to be translated into target code too. There are three prescribed ways in which ALF execution semantics may be implemented, namely *Interpretive Execution*, *Compilative Execution*,

and *Translational Execution*. In this work, the challenge is to provide *translational execution* of ALF through mechanisms able to transform ALF action code first into intermediate concepts and thereby to a non-UML target language (e.g., C++). This choice was dictated by the overall goal of the code generation to produce a non-UML target language.

Research Challenge 3 (RC3) – Define and implement an automated process to enable the back-propagation of monitoring results to design models
The challenge identified in achieving back-propagation is two-fold:

- *Monitoring results and traceability information management*: results coming from the monitored execution of the generated code are part of the source artefacts for back-propagation to the design models; the representation format of this information is pivotal. Monitoring results need to be manipulated in order to extract the observed values and store them in formal structures to be fed to the back-propagating transformations;

- *Annotation of design models*: the very final step of the approach would be the actual enrichment of the design models with values gathered during code execution monitoring activities. The enrichment should be performed by injecting the observed values into the related model elements' placeholders at modelling level according to the mappings contained in the traceability links.

Research Challenge 4 (RC4) – Demonstrate how the round-trip approach can be employed to guide engineering decisions based on back-propagated EFP values
The round-trip approach is meant to be employed as a support for the engineer to take extra-functionally aware decisions at modelling level exploiting values gathered at runtime. The approach provides synthesis of design models to highly resource efficient (in terms of inter-system communications) single process applications or exploit multi process configurations. These different deployment options raise the opportunity of demonstrating how the round-trip approach can be employed by the engineer to exploit measurements gathered at system implementation level for deployment assessment at modelling level. As part of the demonstration, the approach is applied to an industrial case-study in the telecommunication domain.

1.3 Thesis Contributions

The contribution of this research work is represented by an automated round-trip approach for the preservation of system properties throughout the development of embedded systems. Specific novel contributions are:

Thesis Contribution 1 (TC1) – Definition of semantic rules for the exploitation of UML semantic variation points regarding explicit components interconnections via ports and their automated generation
This contribution provides a solution for the automatic generation of explicit component instances and establishment of links among them according to the involved components' and ports' multiplicity, in the structural model of the system. To achieve this, a set of rules have been defined as semantic interpretation of the UML metamodel by exploiting the semantic variation points mechanism provided along with UML. The details about this contribution in terms of the defined semantic rules as well as the model transformation mechanisms implemented for enabling the generation of explicit component instances and links are presented in Chapter 5. This contribution provides a solution for Challenge RC1.

Thesis Contribution 2 (TC2) – Generation of full functional code in C++ from design models to either a single process or to a set of communicating processes for Linux (only single process) and OSE
This contribution enables fully automatic generation of 100% functional C++ code from the design models. The entailed target platforms are Linux and OSE while the design models conform to the CHESS-ML. The description of the various model manipulations (in terms of M2M and M2T transformations) needed to generate traceability links as well as the implementation are described in Chapters 6, 7, and 8, together with the employment of the intermediate modelling artefacts introduced in Chapter 2 and employed to support generation and back-propagation phases. This contribution provides a solution for Challenge RC1.

Thesis Contribution 3 (TC3) – Translation mechanisms for the transformation of ALF concepts to a generic object-oriented intermediate format and thereby to C++
In order to generate 100% functional code, the solution has to be able to transform complex behaviours, defined in the models through ALF, to the target language. The ALF specification proposes three different mechanisms for the ex-

ecution of ALF text; this contribution provides a solution for the *translational execution* of ALF, meant as the translation of the ALF text into a non-UML target language to be executed on a non-UML target platform. The details of the contribution in terms of entailed intermediate modelling artefacts (introduced in Chapter 2) and implementation of the model transformations that perform the translation from ALF concepts to a non-UML target language (i.e., C++) are provided in Chapter 7. While part of the Contribution 2, this solution is considered as a standalone novel contribution agnostic of the considered modelling language and therefore can be employed in any development process in need of a translator from ALF concepts to C++. This contribution provides a solution for Challenge RC2.

Thesis Contribution 4 (TC4) – Provision of back-propagation facilities to correctly inject monitoring results to the appropriate placeholders in the design models.
One of the pillars of this research work is the introduction, in an MDE process, of the novel step of back-propagation to the design models of the extra-functional values gathered at runtime by monitoring the execution of the generated code. In Chapter 9 we describe the features employed for monitoring the code execution on Linux and OSE and gathering extra-functional values at function level. Moreover, we depict text and model manipulations needed to perform the injection of extra-functional values to the design models. This contribution provides a solution for Challenge RC3.

Thesis Contribution 5 (TC5) – Possibility to employ the round-trip approach for deployment assessment activities at modelling level based on measurements gathered at system implementation level
Based on the possibility provided by the solution to entail the generation of multi process applications, we propose a possible employment of the round-trip approach to help the engineer in taking deployment decisions at modelling level based on values gathered at system implementation level on OSE. Details about this contribution are provided in Chapter 9. This contribution provides a solution for Challenge RC4.

1.4 Research Method

This research work has been carried out by following the deductive-like method depicted in Figure 1.1. As first task we studied the literature to identify open

research challenges in the fields of both MDE and CBSE defining context and focus on complex (embedded) systems. During this phase we identified the problem of system (especially extra-functional) properties preservation from models to code as common challenge in the two areas.

Afterwards we defined our research agenda by iterating the following steps: (i) definition of the research problem, (ii) systematic literature review for rounding off the problem, and (iii) definition of research-driving challenges. While the research problem, namely the preservation of system properties from models to code, was quite clear, the involved challenges were not. In order to be able to preserve properties at code level, the implementation should have been completely generated in an automatic manner. Moreover, in order to reach the needed level of details at modelling level to generate 100% of code, advanced means to model behavioural descriptions would have been needed. This led to our choice of employing ALF as action language within the models; this decision added the additional challenge of producing translational mechanisms from ALF to the target language. Additionally, in order to aid preservation in the embedded domain, where some EFPs (e.g., performance) often only emerge in a running product, we introduced the novel concept of back-propagation of monitoring results from the code execution to design models.

Once the research objectives and challenges were finalised, we carried out the development of the solution and gradually presented the results at international forums. These tasks have been performed by iteratively observing, analysing, evaluating and refining the research results. Moreover, in this phase we identified additional challenges that were added to the ones arisen at research definition time. An example of additional challenges was the need of exploiting UML's semantic variation points to generate explicit component instances in order to be able to generate a running application with no need of manual intervention.

To close the circle, we validated our research results by implementing a prototype and testing it against a set of in-house case-studies as well as an industrial case-study in the telecommunication domain; multiple iterations among this and the previous phases were needed to achieve the final solution.

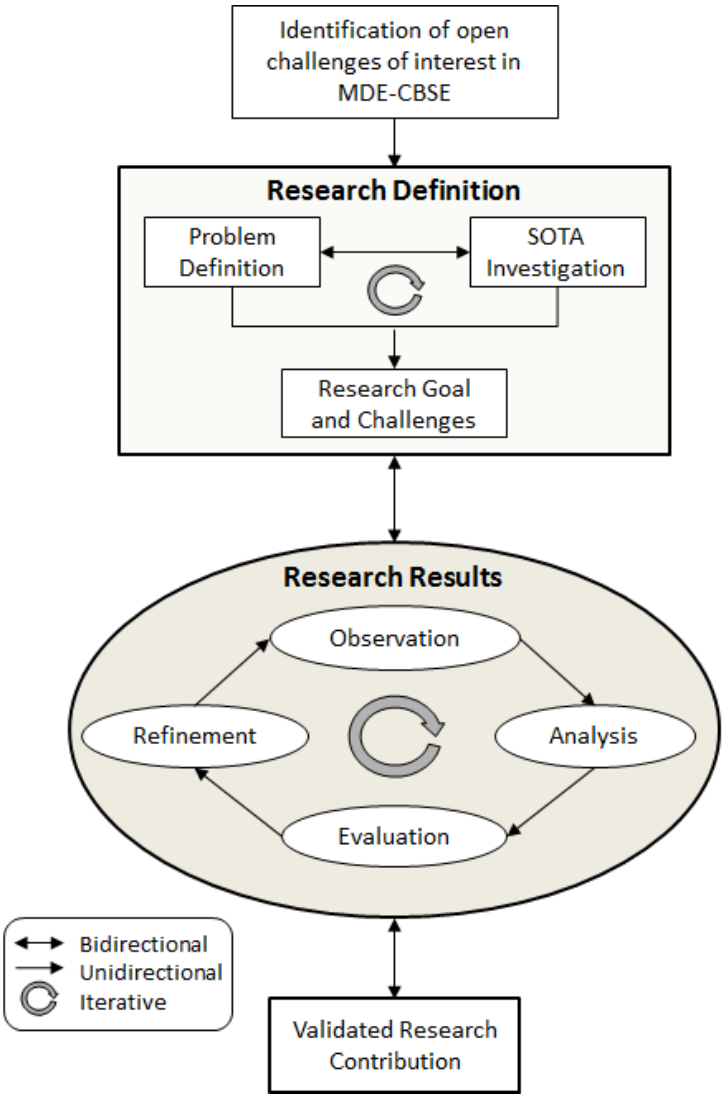


Figure 1.1: Research methodology

1.5 Thesis Outline

In this chapter we describe the outline of the chapters composing this thesis. An overview of the relation between contributions and research challenges as well as describing chapters is summarised in Figure 1.2.

Chapter 1 – Introduction
This chapter introduces the motivation of this research work as well as the definition of the basic concepts we employ throughout the thesis. Moreover, the research settings are described in terms of goal, related challenges and method.

Chapter 2 – Core Artefacts
In this chapter we introduce the auxiliary modelling artefacts (i.e., instance, intermediate, back-propagation metamodels) we defined in order to support the round-trip approach both *forward*, for code generation purposes, and *backward*, for back-propagation needs.

Chapter 3 – Round-trip Approach for Model-driven Development of Embedded Systems: an Overview
It presents an overall description of the round-trip approach in terms of the main steps and artefacts involved, while leaving details to the specific chapters.

Chapter 4 – A Running Example: the AAL2 Subsystem
A running example by means of a re-elaborated industrial system is presented in this chapter. The example will be employed to show the actual application of each of the approach's steps and has been utilised for validation purposes too.

Chapter 5 – Exploiting UML Semantic Variation Points to Generate Explicit Component Instances
In this chapter we describe our solution for the automatic generation of explicit component instances and establishment of links among them according to components' and ports' multiplicity in the structural model of the system. More specifically we depict the set of semantic rules that we defined as semantic interpretation of the UML metamodel by exploiting the semantic variation points mechanism provided along with UML, as well as the actual generation of instances in terms of M2M transformations. The solution is applied to the running example introduced in Chapter 4.

Chapter 6 – Generating Intermediate Concepts
It depicts the generation step where intermediate concepts and traceability links are created starting from the design models. This step focuses on the structural description of the system under development and is achieved through a set of M2M transformations. The application of the mechanism to the running example is provided too.

Chapter 7 – Completing Intermediate Model with Behavioural Descriptions in ALF
This chapter describes the solution for the translational execution of ALF code. More specifically, in-place M2M transformations are defined for translating ALF code to intermediate concepts and injecting them into the already created intermediate model. Examples of translation from ALF to intermediate concepts are given in terms of the running example.

Chapter 8 – Generating Full-fledged C++ from Intermediate Model
The final generation step consisting of the translation from intermediate concepts to C++ is described in terms of the related M2T transformation and its application to the running example.

Chapter 9 – Code Execution Monitoring and Back-propagation
In this chapter we explore the monitoring features that enable the gathering of performance measurements during code execution both on Linux and OSE as well as the model transformation mechanisms (T2M and M2M) providing back-propagation of those measurements to the design models for preservation assessment. Moreover, the usefulness of the proposed round-trip approach is shown through its employment towards deployment assessment in case of multi process applications on OSE.

Chapter 10 – Validation
The validation of the proposed round-trip approach against both in-house case studies as well as in industrial settings is discussed in this chapter. Moreover, details on complexity, scalability and reusability of the approach as a whole as well as step-wise are presented too.

Chapter 11 – Discussion
In this chapter a discussion of the thesis contributions is provided. The focus is reviewing each of the contributions against the related research challenge and highlight both positive aspects and limitations as well as the reasons behind

them. Moreover, a discussion on more general issues is given together with possible solutions.

Chapter 12 – Conclusions
The contents of the thesis are summarised and possible future directions are presented in this chapter.

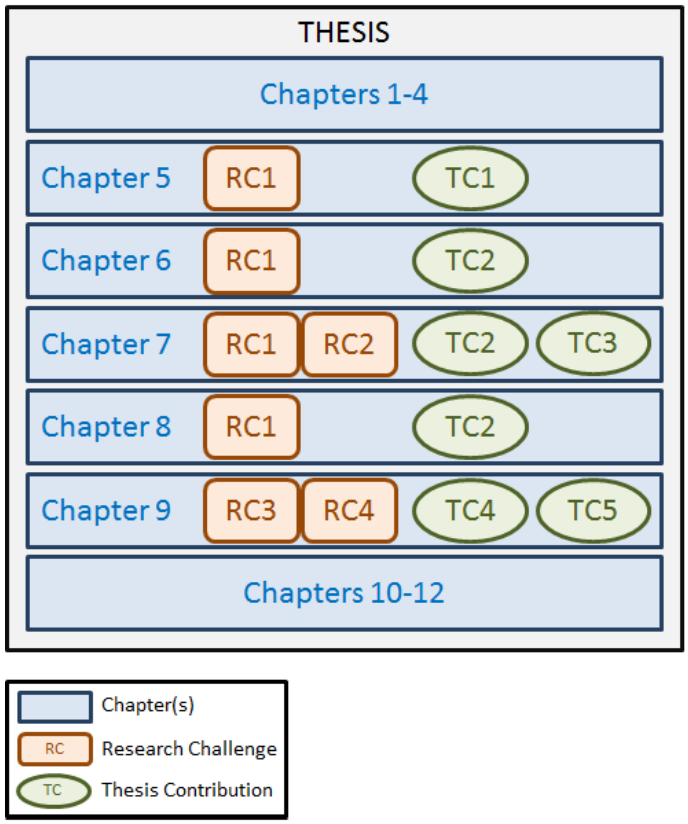


Figure 1.2: Distribution of research challenges and thesis contributions

Chapter 2

Core Artefacts

In this chapter we introduce the intermediate artefacts, in terms of metamodels, that we defined in order to support the round-trip approach. A dedicated section is provided for each of the artefacts.

2.1 Instance Metamodel

The explicit instances of components and ports as well as the links between them need to be stored in properly defined structures used during the generation process but even for eventual model-based analysis and simulation. Since we aim at providing a model-driven development process, the most suitable way to store information concerning model elements is to use models [1]. For performance reasons, namely speeding up the code generation process, we defined an *Instance Metamodel* (InstanceMM) using Ecore (Figure 2.1) in the Eclipse Modeling Framework (EMF) [19] for this purpose. Alternatively, this information could be stored by means of instance specifications available in UML (e.g., object diagram [25]); this solution would be preferable in case of model-based analysis and simulation based on UML. Considering a component in ChessMM, its instances in InstanceMM are represented by the `Component` metaclass; the attributes `mult` and `rel_id` represent respectively the component multiplicity and the relative identifier of the instance in the range $[1, mult]$. Additionally, support for modelling the allocation of component instances to specific processes is provided by means of the attribute `deployTo`.

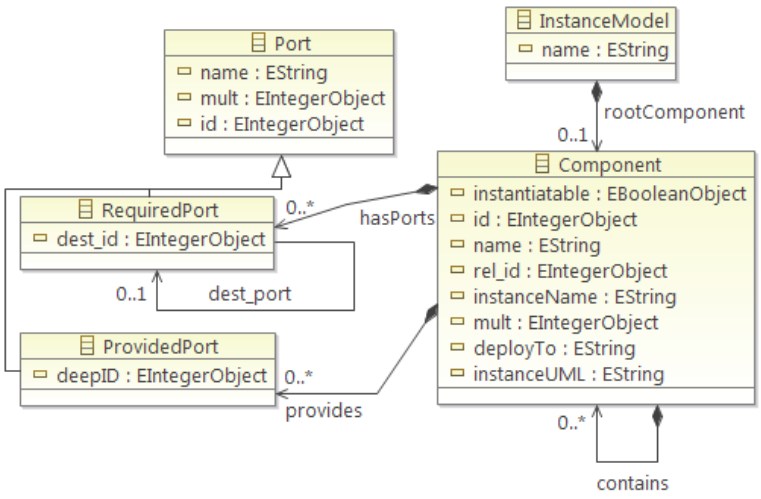


Figure 2.1: Instance metamodel in Ecore

Required and provided port instances are modelled by, respectively, `RequiredPort` and `ProvidedPort` that specialise the abstract metaclass `Port`. Their multiplicity is modelled through `Port`'s attribute `mult`. The relative identifier of a the port instance, in the range $[1, mult]$, is represented by `Port`'s attribute `id`.

2.2 Intermediate Metamodel

The *Intermediate Metamodel* (InterMM) has been defined in Ecore to provide the means for translating the design model to a neutral and more generic object-oriented (OO) abstraction. The usefulness of intermediate artefacts resides in the generic goals of the generation approach, such as domain-independence and reusability, as well as for validation reasons [26]. While a direct generation from design models to C++ would entail an overly complex M2T transformation and could hardly be reused in the generation of other languages, the employment of InterMM allows to split up the generation into a set of simpler M2M and M2T transformations. Doing so, the process is carried out through smaller and less complex steps thus reducing its error-proneness and allowing to focus only on the M2T rules for enabling the generation of other target lan-

guages; the M2M transformations, which represent the most complex steps, would remain largely valid.

Moreover, while the notion of intermediate metamodel in a code generation process is not new (as discussed in Chapter 6), the ability of InterMM to host action code in a modelling fashion at the same abstraction level of the surrounding modelling elements contributes to its originality. In fact, while usually action code is defined in terms of target languages (as in [27, 28]) and therefore taken as it is from the model and injected into generated structural code, we define it at modelling level using proper modelling artefacts (i.e., ALF) and therefore we need to account it into the transformation process and consequently into InterMM.

A graphical definition of InterMM is depicted in Appendix A (due to its size) where we can distinguish three main portions that are used to model the followings:

- **Structure**: identified by the `SubSystem` metaclass which represents the root of any intermediate model and its contained metaclasses;
- **Statements**: identified by the abstract `Statement` metaclass and the specialising metaclasses that represent the different types of statement;
- **Expressions**: identified by the abstract `Expression` metaclass and the specialising metaclasses that represent the different expression kinds.

According to this decomposition of InterMM, in the next paragraphs we give an overview on the main metaclasses composing them.

Structure. This portion of InterMM is used to describe the structure of the system (see Figure A.1 in Appendix A). It addresses the specification of application specific types, classes, interfaces, operations and processes. Moreover it provides constructs for modelling controlled code injection points (i.e., markers) that are exploited for static and runtime code analysis [29]. The main metaclasses are:

- `SubSystem`: represents the root model entity and it is composed by processes, types, global variables, structures, defines, functions and references to external functions;
- `Process`: represents the processes on which the component instances are deployed;

- `Operation`: models operations and consists of a signature and a set of statements;
- `ScopedNamedInstance`: represents a named accessible entity that may represent, e.g., component's attributes/properties or variables defined in the behavioural specifications by means of, e.g., an action language like ALF;
- `Type`: is the abstract metaclass which is specialised by a set of metaclasses defining the different types that can be modelled in the modelling language (e.g., primitive types, enumerations, redefined types, complex types) together with code-related types (e.g., references, garbage collector references);
- `Class`: it specialises `Type` and owns a set of methods and attributes. It can represent, e.g., component instances, state-machine states and complex datatypes;
- `InjectionMarker`: it is a metaclass that can be employed to model injection markers to be put in the code in a controlled manner for, e.g., analysis purposes. The use of code injections has to be handled carefully also at M2T transformation level in order not to jeopardise the implementation.

Statements. The body of operations, if specified through action code, would be composed by a sequentially ordered set of statements. This portion of InterMM is devoted to the definition of the concepts needed to model statements (see Figure A.2 in Appendix A). The main metaclasses are:

- `Statement`: is the core abstract metaclass;
- `SimpleInvocation`: represents the invocation of a function and it contains an `Invocation` expression (see next paragraph);
- `Assign`: represents the assignment of a value to a variable. It contains a specific `InstanceAccess` expression (see next paragraph);
- `Control`: is the abstract metaclass representing statements related to the control flow of the application. It is specialised by specific metaclasses such as: `If`, for modelling *if* loops and *switch* cases, `ForRange`, `ForEver` and `ForEach`, for modelling the various different *for* loops available in ALF, `While`, for modelling *while* loops, and `Return`, for

modelling the exit from an operation body. If the operation has a defined return-type, the statement includes an expression evaluating it;

- `InjectionStatement` represents the statement that can be introduced in the code through injection markers;
- `Inline`: it is used to place inline statically defined statements in the generated code. It should be employed mainly for debugging and analysis purposes (e.g., check the correctness of the control flow when testing the application) and hence be avoided when generating the final implementation version.

Expressions. This portion of InterMM provides the means to define expressions, meant as behavioural units that evaluate a collection of values (see Figure A.3 in Appendix A). The main metaclasses are:

- `Expression`: is the core abstract metaclass;
- `Invocation`: represents the expression related to the invocation of a function and defines input parameters and expected return type;
- `RTTI`: represents an abstract metaclass that is specialised by metaclasses, such as `Instanceof` and `Hastype`, for defining type introspection if provided by the modelling or action language (as in the case of ALF);
- `Alloc`: is a metaclass used to define a memory allocation in case the target programming languages would require it;
- `Define`: represents the specification of symbolic constant declarations (i.e., `#DEFINE` or similar, depending on the target programming language);
- `InstanceAccess`: is an abstract metaclass for the definition of access to variables;
- `ReferenceAccess`: represents the access to a variable's reference;
- `ValueAccess`: represents the access to the value of a variable;
- `Literal`: is an abstract metaclass representing the different constant literal values. It is specialised by metaclasses such as `IntegerValue`, `StringValue`, `BooleanValue`, that define the specific literal type;

- `Computation`: represents an abstract metaclass which is specialised by `Unary` and `Binary` respectively defining an expression that executes a unary operator and an expression that executes a binary operator.

InterMM is meant to accommodate a wide range of concepts that are then interpreted in a specific way by the M2T transformation depending on the features provided by the target programming languages. This means that, while the syntax is fixed, the semantics that the various metaconcepts assume might change from one target programming language to another and is therefore embedded in the target-specific M2T transformation.

2.3 Back-propagation Metamodel

In the proposed approach, the task of automating the generation of implementation code does not only concern the actual transformation from design models to code since tracing information between model elements and generated code has also to be defined for enabling back-propagation activities. Traceability can be any relationship existing between artefacts within a software engineering life cycle. These relationships include: (i) explicit links derived from for/backward transformations, (ii) links derived from code analysis, (iii) inferred links computed on the basis of change management of system's items [30]. In our work we consider explicit links derived from transformations as relationships between artefacts.

Since our approach relies on models, transformations and generated code as main artefacts, definition and maintenance of traceability links to cope with consistency among them are crucial. That is the reason for which model transformations in charge of code generation shall be properly defined by encoding apposite rules for the generation of explicit traceability links between source and target. In this way, information exchanged among models through transformations is formally stored and maintained in structures that are easily and univocally navigable and pieces of information reachable following precise patterns.

For this purpose we defined the *Back-propagation Metamodel* (BackMM), depicted in Figure 2.2, using Ecore; a slightly similar structure for storing tracing information can be found in [31]. BackMM has been defined for enabling the creation of back-propagation models (BackM) which store the information gathered during code generation and monitoring tasks in a structured manner.

Conceptually, two classes of information are stored in BackMM: (i) traceability information and (ii) monitoring results. Traceability information is

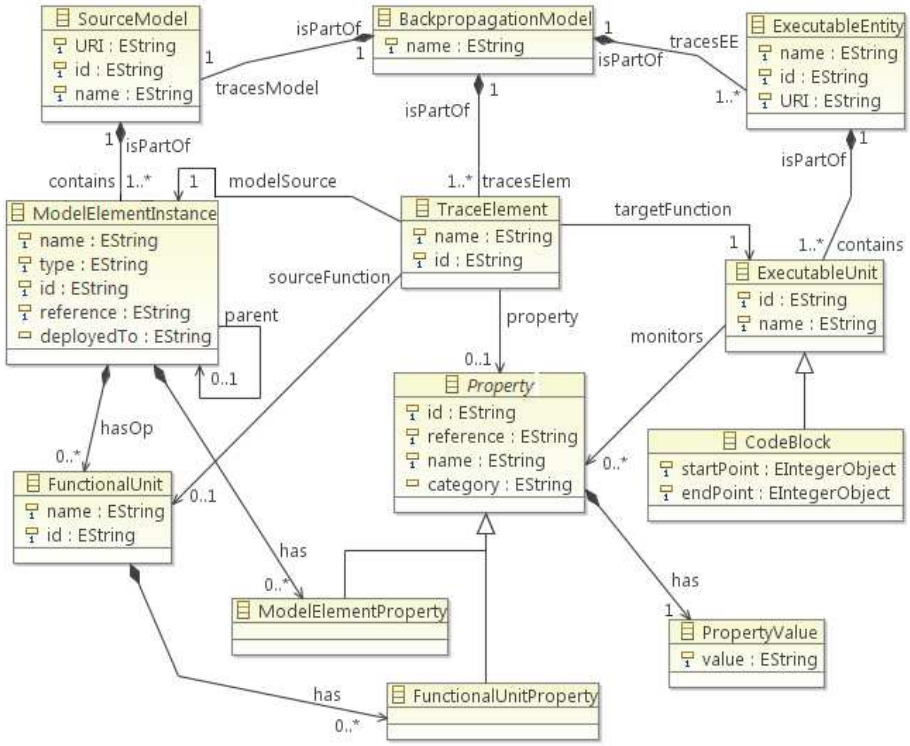


Figure 2.2: Back-propagation metamodel

composed by explicit trace links created during the code generation task and stored in terms of trace elements between model elements and code. The core concept in BackMM is indeed the *trace element*, which allows the navigation through all the stored information needed for back-propagating the observed values to the design models. A trace element *TE* (TraceElement in Figure 2.2) can store traceability at two different granularities:

- **Model element level**: in this case it is represented as a triple <*ME, EU, MEP*> where *ME* is a model element (`ModelElement` in Figure 2.2) contained in a design model *SM* (`SourceModel` in Figure 2.2), *EU* is an executable unit (`ExecutableUnit` in Figure 2.2) contained in an executable entity *EE* (`ExecutableEntity` in Figure 2.2), and *MEP*

(`ModelElementProperty` in Figure 2.2) is an EFP defined for *ME* and calculated by monitoring the execution of *EU*. A typical case for this granularity is the component in a component-based architecture;

- **Model element's functional unit level**: in this case it is represented as a quadruple <*ME, FU, EU, FUP*> where *ME* and *EU* represent the same information as for the model element level. The further level of granularity is maintained by *FU* which represents an operation/method (`FunctionalUnit` in Figure 2.2) defined in the model element specification *ME* and *FUP* (`FunctionalUnitProperty` in Figure 2.2) which represents an EFP defined for *FU* and meant to be calculated by monitoring the execution of *EU*. A typical case for such a granularity is the component's operation in a component-based architecture.

More generally, BackM (`BackpropagationModel` in Figure 2.2) is a triple <*TE*, SM, EE**>, where *TE** is a non-empty set of trace elements, *SM* is a design model (i.e., a composition of model elements *ME* and functional units *FU*), and *EE** is a non-empty set of executable entities which are in turn composed by executable units *EU*. Depending from the code generation and the monitoring activities, an executable unit could be defined in a more detailed manner as a code block with start and end point within the code file (i.e., the executable entity).

Apart from the traceability links, BackM hosts the information extrapolated from the monitoring activities. More specifically, each property *P* defined in BackM has a property value *V* (`PropertyValue` in Figure 2.2), which is calculated during monitoring activities and represents the value to be propagated back to the related extra-functional annotation's placeholder in the design model.

2.4 Summary

In this chapter we introduced three intermediate artefacts, namely instance metamodel, intermediate metamodel and back-propagation metamodel. The instance metamodel hosts information regarding component instances and explicit links between them which are generated from the initial design model. The intermediate metamodel is thought to be the core intermediate artefact to which all the information, both structural and behavioural, coming from the design model is translated to and from which the further generation of code

towards various platforms is performed. Finally, the back-propagation meta-model stores information regarding explicit traceability links created during the code generation phase and it is utilised as bridge between monitoring results and design model for back-propagation activities.

Chapter 3

Round-trip Approach for Model-driven Development of Embedded Systems: an Overview

The overall contribution of the research work presented in this thesis is represented by a full model-driven approach for the development of embedded systems. The focus is on the provision of automatic mechanisms to exploit measurements gathered at system implementation level for assessment of system properties preservation. In order to achieve such an approach, depicted in Figure 3.1, two main steps had to be provided: (1) *forward*, meaning automatic generation of full-fledged functional code from the design models, and (2) *backward*, through monitoring of code execution and back-propagation of observed values to the design models.

Both steps consist of a number of substeps that we describe in the following paragraphs. In the remainder of the thesis we employ these abbreviations when describing modelling artefacts within model transformations and algorithms:

- UmlMM: UML metamodel
- UmlM: UML model
- ChessMM: CHESS metamodel
- ChessM: CHESS model
- AlfMM: ALF metamodel

- AlfOpM: ALF operation model
- InterMM: intermediate metamodel
- InterM: intermediate model
- InstanceMM: instance metamodel
- InstanceM: instance model
- BackMM: back-propagation metamodel
- BackM: back-propagation model

Automatic generation of code. Starting from the design of the system under development defined as ChessM, the first step is represented by the generation of component instances and explicit links according to components' and ports' multiplicity defined in ChessM (Figure 3.1.a). This information is stored in terms of InstanceM, conforming to the previously defined InstanceMM. The generation is driven by a set of semantic rules we defined to fix the related semantic variation point in the UmlMM. Practically, the generation is achieved through a M2M transformation taking in input ChessM and providing InstanceM as output.

A further M2M transformation takes in input ChessM and the newly created InstanceM for translating the structural description of the system into intermediate concepts (Figure 3.1.b). The outcome of this transformation is InterM, that conforms to InterMM. Moreover, during this task, explicit traceability links are created (Figure3.1.c) in terms of BackM, conforming to BackMM, and will be used for driving the back-propagation phase.

InterM is then completed with information regarding the behavioural specification of the system defined by means of ALF. This is achieved by: parsing each ALF operation in the system (Figure 3.1.d) and then run an in-place M2M transformation (Figure 3.1.e) that translates ALF to intermediate concepts and injects them into InterM. Finally the executable C++ implementation is generated through M2T transformations taking as input the sole InterM (Figure 3.1.f).

Monitoring and back-propagation. Once code and traceability links have been generated, EFPs can be evaluated by specific code execution monitoring routines (Figure 3.1.g). Depending on their output format, different actions varying from T2M to M2M transformations (Figure 3.1.h) are required to extract and formalise gathered values, which are thereby injected in the BackM via an in-place T2M transformation in order to have a complete link from models to values. The last step of the round-trip approach annotates ChessM with

the monitoring values (Figure 3.1.i) through a dedicated in-place M2M transformation.

At this point ChessM can be evaluated by means of actual extra-functional values and, if needed, it can be tuned by the developer (e.g., by changing the allocation of components to processing units) to generate more resource-efficient code (Figure 3.1.k). Thanks to the automated support, the process can be reiterated at will until the developer is satisfied with the generated code.

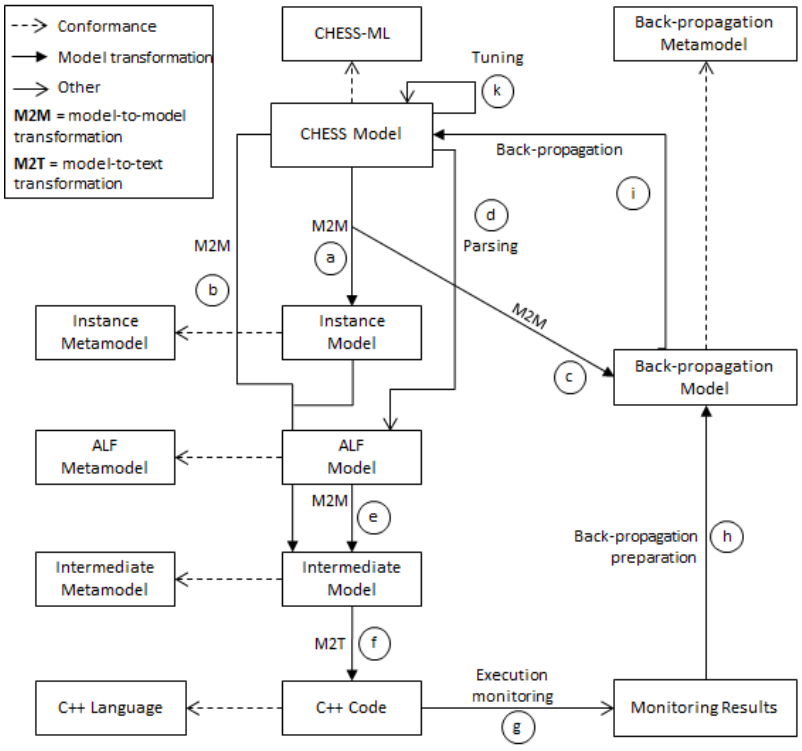


Figure 3.1: Research contribution

Chapter 4

A Running Example: the AAL2 Subsystem

The solution proposed in this work has been validated against case-studies, both in-house as well as in industrial settings, modelled using the CHESS-ML. More specifically regarding the industrial case-study, we employed the Asynchronous Transfer Mode (ATM) [32] Adaptation Layer 2 (AAL2) subsystem, originally intended to adapt voice for transmission over ATM and currently used in telecommunications as part of connectivity platform systems.

The AAL2 subsystem is described in this chapter and utilised as running example for showing each of the steps composing the round-trip approach. For confidentiality reasons, the system models presented here are a re-elaboration of the actual models. Moreover, since the actual ChessM representing the AAL2 subsystem was composed by several hundred thousands of component instances and multiple levels of hierarchical composition of components, we employ a simplified version as running example for simplicity reasons. Anyhow, in Chapter 10 we provide all the details regarding the evaluation of the proposed approach on the complete AAL2 model.

In Figure 4.1 we propose the simplified AAL2 subsystem (i.e., `SwSystem` composite component) which is composed by three main components: (i) `NCC`, (ii) `AAL2RI_Client`, (iii) `NCIClient`. Each of these components has a complex internal structure in terms of composition of other components; in this example we only show part of the `NCC` internal structure while considering `AAL2RI_Client` and `NCIClient` as black-boxes. `NCC` is a connections handler providing connectivity services for the establishment/release

of communication paths between pairs of connection endpoints handled by AAL2RI_Client. NCIClient represents an application asking for services provided by NCC and its underlying layers; the components communicate through functional interfaces (function calls or message passing depending on the deployment configuration) exposed by their provided ports.

The NCC component has a complex internal structure (Figure 4.1) composed of the following components: NodeConnHandler, which dispatches the incoming connection requests to available NetConn instances, NetConn, that controls establishment and release of network connections between nodes (NodeConneElem instances), NodeConnElem, that handles management of connections to the network within the single node, and PortHandler, which manages connection resources. Each of these subcomponents has in turn a complex internal structure in term of components composition; in this case-study we consider only the first two levels of decomposition (down to the NCC's internal structure).

The behavioural definition of the system is given by means of UML state-machines enriched with action code definitions for the involved operations specified by means of ALF. In Figure 4.2 the AAL2's NodeConn state-machine is depicted together with the ALF code specifying the behaviour of the operation NodeConn_riDisconnectCfm. A typical connection scenario in the AAL2 subsystem is the establishment of a connection between two endpoints residing on the same node. This is a constrained case of a more general network-wide connection where the two end-points reside on different nodes and the communication transits through a number of other intermediate nodes in the network. When NCIClient wants to connect two end-points, a connection setup request is sent to NCC through the PI_NCI_2_NCC interface; this request contains information about the end-points. NCC asks for the establishment of a connection segment between the end-points to an external component (not modelled in this case-study). Then it sends a request through the RI_NCC_2_AAL2RI interface for each end-point to their respective AAL2RI_Client to activate the access to the transport layer. Once both end-points have positively responded through their RI_AAL2RI_2_NCC interface, NCC confirms the establishment of the connection to NCIClient through the RI_NCC_2_NCI interface.

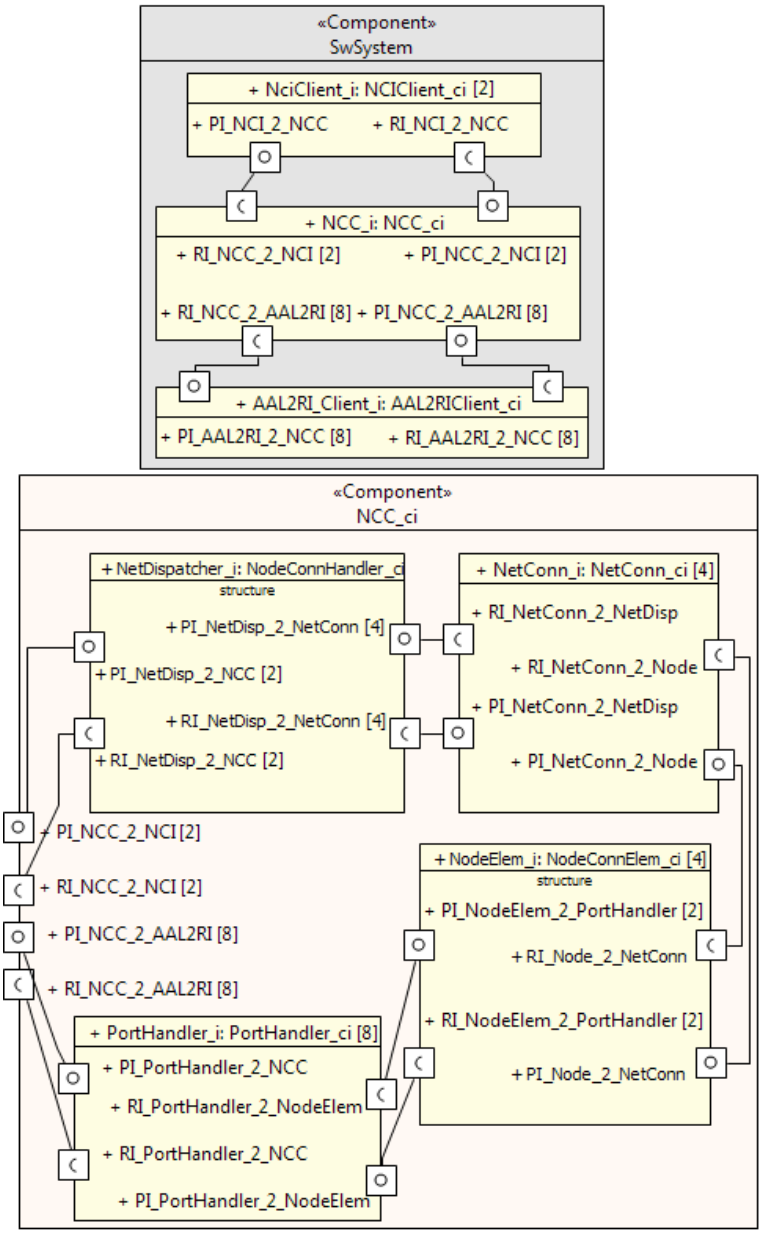
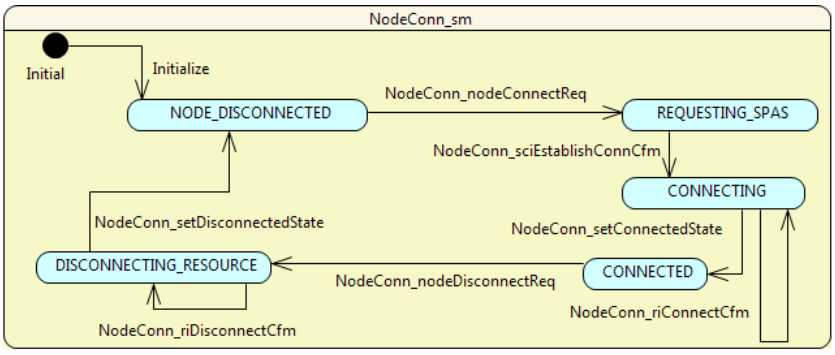


Figure 4.1: AAL2 subsystem structural design in CHESS




```
public NodeConn_riDisconnectCfm ( in portId : model::modelComponentView::DataTypes::Cello_Port,
in serverConn_r : model::modelComponentView::DataTypes::Cello_RiServerConn)
{
/*@inline('C++')
   "ENTER( \"NodeConn_riDisconnectCfm. \" );"
*/
if(serverConn_r.serverConnId == serverConnId && connHalf[serverConn_r.connHalf].portId == portId)
{
   connHalf[serverConn_r.connHalf].respondState =
        model::modelComponentView::DataTypes::NCCdataTypes::RespondState[RI_RESPONDED_CFM];

   model::modelComponentView::DataTypes::U8 secondHalf =
                                          NodeConn_getSecondHalf(serverConn_r.connHalf);
   if(connHalf[secondHalf].respondState !=
  model::modelComponentView::DataTypes::NCCdataTypes::RespondState[WATING_FOR_RESPONSE])
   {
      model::modelComponentView::DataTypes::NCCdataTypes::SciState x =
                                          SciHandler_RI.SciHandler_getSciConnectionState();
      if(x == model::modelComponentView::DataTypes::NCCdataTypes::SciState[SCI_ATTACHED])
      {
         NodeConn_setDisconnectingSpasState();
         SciHandler_RI.SciHandler_sciReleaseConnReq(spasConnId, serverConnId);
         // Simulate  NodeConn_sciReleaseConnCfm();
      }
      else
      {
         NodeConn_setDisconnectedState();
         NetConn_RI.NetConn_nodeDisconnectCfm();
         NodeConn_removeData();
      } } }
```

Figure 4.2: `NodeConn` state-machine and `NodeConn_riDisconnect-Cfm` operation in ALF

Chapter 5

Exploiting UML Semantic Variation Points to Generate Explicit Component Instances

The UML allows the specification of a system in a component-based fashion as assembly of components communicating via required and provided interfaces (exposed by ports). As stated in the language specification, a port represents an interaction between a classifier (i.e., component) instance and its internal or external environment. Additionally, features owned by required interfaces are meant to be offered by one or more instances of the owning classifier to one or more instances of the classifiers in its internal or external environment [10].

In other words, multiple instances of components can provide features via ports to multiple instances of both peer (at the same hierarchical level) and internal components. While the number of instances of components and ports can be precisely specified, the port-to-port links are not equipped with a detailed specification of the component instances they connect. In fact, the links are defined at classifier level thus on the sets of instances and the UML metamodel leaves the rules for matching the multiplicities of connected sets of instances (components) and ports as a *semantic variation point* [10]. In order to be able to automatically generate the set of actual links between explicit component instances the solution cannot prescind from defining the semantics needed to

fix the related variation point.

As depicted on the left-hand side in Figure 5.1, the two components (or sets of instances since their multiplicity is greater than 1) A and B are linked through a single connector that does not carry any information on the actual instance-to-instance connection. This information is crucial to be able to properly analyse important properties of the system at modelling level, enable model simulation and generate executable code. That is why semantic rules should be specified for generating the actual links among explicit component instances. Alternatively, this information could be manually modelled in different ways (e.g., ad-hoc stereotyped annotations, OCL constraints) or created on-the-fly (e.g., through action languages), but, when dealing with complex systems consisting of thousands of component instances, the manual effort demanded by this kind of activity is overwhelming. The same information could even be manually described at code level; different complications, such as inconsistency between models and code, unintentional injection of errors in the code as well as large effort in carrying out the manual coding task arise thus making this solution simply infeasible in developing industrial systems. For these reasons, automatic generation is preferable in order to generally mitigate the developing team's effort demanded by modelling activities.

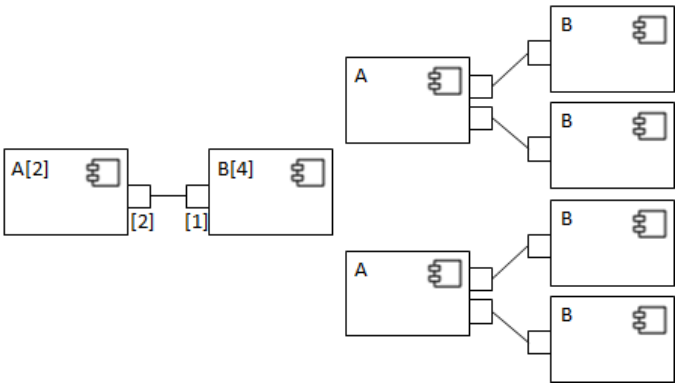


Figure 5.1: UML-like component instances and links ([x] = multiplicity)

In defining the generation of links between instances we need to take into account the fact that UML allows the definition of hierarchical composite components with unlimited depth levels. In fact, links do not always concern component instances placed at the same hierarchical level but even instances defined

at different depths in the composition structure. Especially when composition hierarchy does not imply delegation of the provided features by internal classifiers to the container classifier, the ability to generate explicit links between component instances placed at different hierarchical levels becomes crucial for increasing model analysability and providing complete code generation capabilities.

In this chapter we explore the solution we provide for the automatic generation of explicit component instances and the establishment of links among them according to the involved components' and ports' multiplicity defined in the structural model of the system. This generation is made possible by a set of rules that we defined as semantic interpretation of the UML metamodel. On the right-hand side in Figure 5.1 the general idea of generating explicit component instances and links is applied to the two sets of instances A and B introduced before. More specifically, the components A and B are unwound by means of explicit instances according to their multiplicity and the actual links among them are created.

5.1 Assumptions

In order to identify the scope in which our solution has been developed and its ultimate goal, a number of assumptions and clarifications must be made. According to the constraints given when defining the CHESS-ML, for ensuring guarantees at runtime of the properties modelled at design time, dynamic instantiation of components is not allowed; that is the reason for which our solution entails only prefixed cardinality on components and ports. For the same reason, multiplicities are defined as concise values (i.e., `[n]`) while range values (e.g., `[n..m]`) are left as possible future enhancement. Moreover, two further assumptions have been made in regards to components interconnections: (i) the connectors linking components via ports have multiplicity 1, thus leaving components' and ports' multiplicity as variables for the interconnections calculation and generation, (ii) only binary connections are considered, leaving n-ary possibilities as future work.

Nonetheless, these assumptions did not prevent us from the ability to model complex industrial systems, but were rather derived to circumscribe the scope of the problem and therefore propose a technically valid solution. Since no unique interpretation of the UML metamodel semantics can be given for a semantic variation point, we focused on the problems to be addressed within our scope with particular attention to full-fledged code generation.

5.2 Definition of Semantic Rules

The guidelines given by the superstructure specification of UML do not provide a fixed semantics concerning the multiplicities of connected components and ports. This means that any combination of multiplicities is syntactically allowed and therefore syntactically correct. In order to be able to give a semantics to the interrelationships among components and to generate explicit links between component instances, boundaries have to be set for their syntactical definition. In other words, only a set of meaningful combinations of multiplicities of connected components are considered in order for the semantic rules to be deterministically applied. Let us refer to the most general case of connection between two components A and B; their multiplicity is expressed respectively as M_A and M_B. An explicit instance of component A is represented by i_A and assumes values in the range $[1, M_A]$. Concerning component B, an explicit instance is represented by i_B and assumes values in the range $[1, M_B]$.

Moreover, component A is connected through its required port A_RP to component B's provided port B_PP; the multiplicities of these ports are expressed respectively as M_{A_RP} and M_{B_PP}. An explicit instance of port A_RP is represented by i_{A_RP} and assumes values in the range $[1, M_{A\_RP}]$. Regarding port B_PP, an explicit instance is represented by i_{B_PP} and assumes values in the range $[1, M_{B\_PP}]$.

The basic rule for the specification of components and ports multiplicities is that: *each required port instance should be connected to one and only one provided port*. According to our general connection case, each i_{A_RP} of each i_A should be connected to one and only one i_B through one i_{B_PP}. Hence the condition in Equation 5.1 shall always be $true$ for the generation process to be able to link an instance i_A, requesting features, to the right instance i_B, providing the features, through their port instances.

$$M_A \times M_{A\_RP} = M_B \times M_{B\_PP} \tag{5.1}$$

At this point, given both the component instance i_A and the instance i_{A_RP} of its required port, we automatically derive the link to the right instance i_B. We defined the following set of five semantic rules for creating such links based on the specified multiplicity combinations (where $\lceil x \rceil$ denotes the ceiling function mapping x to the smallest integer greater than or equal to x):

$$(M_A = M_B) \wedge (M_{A\_RP} = M_{B\_PP}) \Rightarrow i_B = i_A \quad \text{(Rule 1)}$$

$$(M_A \neq M_B) \wedge (M_A = 1) \wedge (M_{A\_RP} = M_B \times M_{B\_PP}) \Rightarrow i_B = \left\lceil \frac{i_{A\_RP}}{M_{B\_PP}} \right\rceil \quad \text{(Rule 2)}$$

$$(M_A \neq M_B) \wedge (M_B = 1) \wedge (M_{B\_PP} = M_A \times M_{A\_RP}) \Rightarrow i_B = 1 \quad \text{(Rule 3)}$$

$$(M_{A\_RP} \neq M_{B\_PP}) \wedge (M_{A\_RP} = 1) \wedge (M_A = M_B \times M_{B\_PP}) \Rightarrow i_B = \left\lceil \frac{i_A}{M_{B\_PP}} \right\rceil \quad \text{(Rule 4)}$$

$$(M_{A\_RP} \neq M_{B\_PP}) \wedge (M_{B\_PP} = 1) \wedge (M_B = M_A \times M_{A\_RP}) \Rightarrow i_B = (i_A \times M_{A\_RP}) - (M_{A\_RP} - i_{A\_RP}) \quad \text{(Rule 5)}$$

Rule 1 represents the situation where A and B have same multiplicity as well as the connecting A_RP and B_PP ports. This is a simple case, in which the indexes of A and B coincide ($i_B = i_A$). Rule 2 takes care of the case in which M_A and M_B are different (with $M_A = 1$) and $M_{A\_RP} = M_B \times M_{B\_PP}$. In this case i_B is calculated through ceiling the quotient of i_{A_RP}/M_{B_PP}. Rule 3 represents the case where M_A and M_B are different (with $M_B = 1$) and and $M_A = M_B \times M_{B\_PP}$. In this case $i_B = 1$ since $M_B = 1$. Rule 4 accounts the case where $M_A = M_B \times M_{B\_PP}$ and A_RP and B_PP ports have different multiplicity (with $M_{A\_RP} = 1$). Here i_B is calculated through ceiling the quotient of i_A/M_{B_PP}. Rule 5 takes care of the case in which which $M_B = M_A \times M_{A\_RP}$ and A_RP and B_PP ports have different multiplicity (with $M_{B\_PP} = 1$). Here i_B is calculated through the expression $(i_A \times M_{A\_RP}) - (M_{A\_RP} - i_{A\_RP})$.

The same rules can be applied in case of delegation among provided or required ports in order to generate links between them; in this case, A_RP and B_PP would both represent provided or required port instances. Note that the defined rules are mutually exclusive therefore there is no fixed order in which they are supposed to be applied.

The considered set of multiplicity combinations does not cover all the possibilities that satisfy Equation 5.1, but rather the set for which no added modelling effort was requested from the developer side. The possibility which is

not covered is represented by Equation 5.2 and represents the case in which Equation 5.1 is satisfied and all the operands are greater than 1.

$$(M_A = M_{B\_PP}) \wedge (M_B = M_{A\_RP}) \wedge (M_A \neq M_B) \tag{5.2}$$

5.3 Relation with Instance Metamodel

Let us generalise the representations introduced when defining the semantic rules in terms of the concepts provided by InstanceMM. Considering component A, the instances i_A are represented by the `Component` metaclass; the attributes `mult` and `rel_id` represent respectively the component multiplicity M_A and the value assumed by i_A in the range $[1, M_A]$. Mirror reasoning applies to component B. Port instances i_{A_RP} and i_{B_PP} are represented by, respectively, `RequiredPort` and `ProvidedPort` that specialise the abstract metaclass `Port`. The multiplicities M_{A_RP} and M_{B_PP} are modelled through `Port`'s attribute `mult`. The value assumed by i_{A_RP} in $[1, M_{A\_RP}]$ and by i_{B_PP} in the range $[1, M_{B\_PP}]$ are represented by `Port`'s attribute `id`.

After generating explicit instances of components and ports in the terms described above, it is possible to apply the defined semantic rules for creating the explicit links between component instances. As aforementioned, given i_A and i_{A_RP}, the idea is to find the right instance i_B to which i_A is connected through i_{A_RP}; once found, this index is stored in the attribute `dest_id` (or `deep_id` if considering provided ports) of i_{A_RP}, thus making i_{A_RP} point to the right instance i_B. In this way, at the end of the process, we will obtain an instance model (InstanceM) conforming to InstanceMM and containing all the explicit instances of components and ports as well as the actual instance-to-instance links.

5.4 Generation Process

The generation of explicit instances and links is achieved through a M2M transformation defined in Operational QVTo [33]. Taking as input a ChessM (or more generally a component-based description of the system defined as a UmlM), the transformation generates the explicit instances of components and ports and then creates the explicit links between component instances by applying the previously defined semantic rules. The multiplicities must be aligned to the rules defined earlier in this section for the generation process to operate correctly. The output of the transformation is an InstanceM.

The transformation has to take into account all the possible connections between ports in a component-based design pattern with unlimited hierarchical levels. These connections can be summarised as follows:

- **Provided to Provided**: in case of composite structures, container and contained component instances can be connected via provided ports for modelling delegation of features' provision visibility to the environment;
- **Required to Required**: similarly, container and contained component instances can be connected via required ports for modelling delegation of features' request to the environment;
- **Required to Provided**: connecting component instances via a link between required and provided port respectively, represents the actual client-server interaction where a component instance owning the required port requires features that the one owning the provided port offers.

The workflow (generalised for UmlMM and valid for ChessMM too) of the transformation is summarised in the pseudo-code shown in Algorithm 1.

Algorithm 1 M2M Transformation from UmlM to InstanceM

$Uml2Instance(in\ UmlM, out\ InstanceM)\{$
for *each component c in* $UmlM$ **do**
  $InstanceM = c.createInstances();$
end for
for *each comp instance* $cInst$ *in* $InstanceM$ **do**
  $cInst.setProv2Prov();$
end for
for *each comp instance* $cInst$ *in* $InstanceM$ **do**
  $cInst.setReq2Prov();$
end for
for *each comp instance* $cInst$ *in* $InstanceM$ **do**
  $cInst.setReq2Req();$
end for
$\}$

The main transformation rules work as follows:

- `createInstances()`: for each component in UmlM, a set of `Component` elements is created in InstanceM; in addition, `ProvidedPort`

and `RequiredPort` elements are created for both provided and required ports of the UML component. The number of component and port instances to be created is, according to CHESS-ML that does not entail value ranges in multiplicities, equally represented by the *lowerBound* and *upperBound* attributes of the related UML element; in our transformation we employ the *upperBound* attribute. Moreover, the hierarchical structure of the components is kept intact in order to correctly generate the links between them;

- `setProv2Prov()`: in UmlM, containing components may be connected to contained components via provided-to-provided port connection. In InstanceM, for each generated component instance, starting from the root[1], we create the explicit links between its provided ports to the component instance owning the provided port at the other end of the connection. The rule is then recursively applied to the contained component instances;

- `setReq2Prov()`: peer components are connected via required-to-provided port connection. In InstanceM, for each generated component instance, starting from the root, we create the explicit links between its required ports to the component instance owning the provided port at the other end of the connection. The rule is then recursively applied to the contained component instances;

- `setReq2Req()`: at this point, each explicit required port instance points to the right component instance owning the provided port instance at the other end of the connection. In UmlM, container components may be connected to contained components via required-to-required port connections. In this case, the transformation sets these missing links in a similar way as for provided ports in `setProv2Prov()`. The rule is then recursively applied to the contained component instances.

Setting links between port instances, regardless of the connection type (i.e. required to provided, provided to provided, required to required), is done by applying the semantic rules we defined. InstanceM generated for the AAL2 subsystem is depicted in a simplified UML-like fashion in Figure 5.2. From the AAL2 model, composed by 7 components, 24 ports and 14 connectors, the transformation mechanism generates an instance model composed by 23 component instances, 136 ports and 80 connectors.

[1] By root component it is meant the one at the root of the hierarchical composition. For peer components we mean components placed at the same hierarchical level.

While the automatic generation of these instances took only few seconds, a manual modelling of each single interconnection instance, besides being error-prone, would not have been as fast; these advantages are amplified when dealing with actual industrial models. In fact, the generation of InstanceM for the complete AAL2 model, composed by 2003 component instances, 14000 port instances and therefore several thousands of connections among their ports, was achieved in around 3 on a conventional laptop.

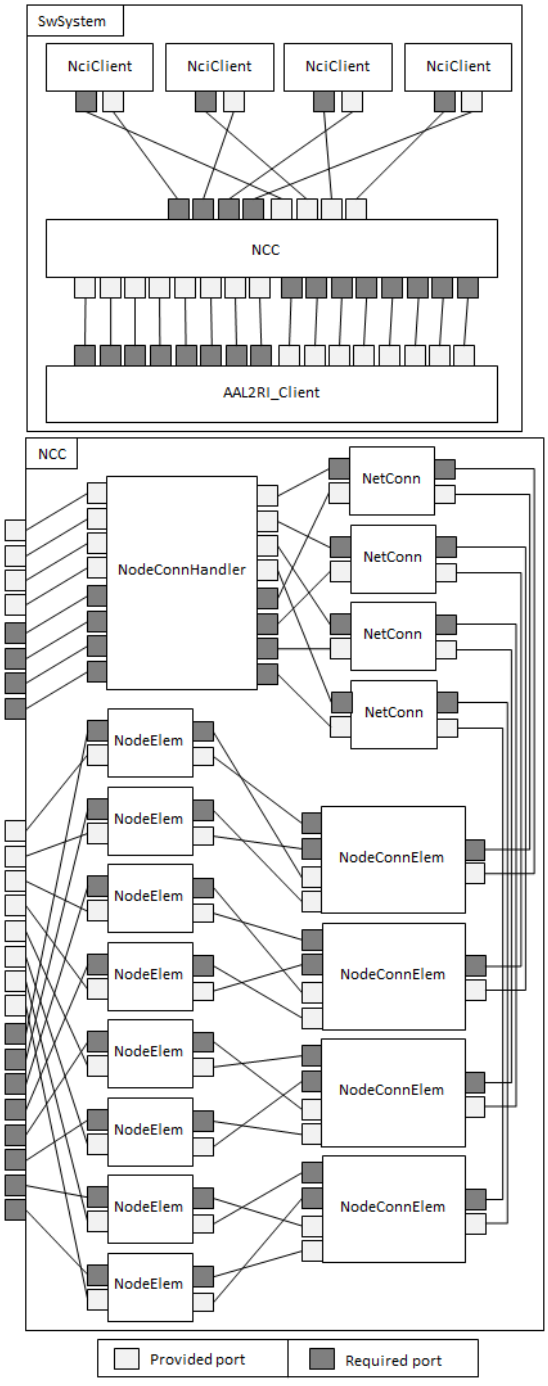


Figure 5.2: InstanceM for the AAL2 subsystem

5.5 Summary and Related Work

In this chapter we described a solution for the automatic generation of explicit component instances and establishment of links among them according to the involved components' and ports' multiplicity defined in the structural model of the system. To enable the automatic generation, we defined a set of rules as semantic interpretation of the variation points defined in the UML metamodel. This represents the first transformation step towards the generation of C++ from ChessM that, taking as input ChessM, provides InstanceM as result; the transformation process is developed using the QVTo transformation language and is packaged in a standalone Eclipse plugin.

The concept of component has been introduced in the UML 2.0, together with an appropriate diagram [10]. In [34] the author discusses composition mechanisms provided with UML 2.0, and in particular the role of multiplicities in interconnections between composite structures. However, it only touches upon the issue of realising component interconnections at different abstraction levels while leaving aside the concrete instantiation.

Generally, several works can be found in the literature addressing issues related to semantic variation points, such as [35] where the authors reify the semantic variation points concerning synchronicity in state-machines towards code generation using the KerMeta meta-language [36] for fixing semantics, or [37] where the authors provide matters to disambiguate variation points related to other aspects of UML diagrams.

Other research works have been devoted to giving semantics across links through ports and keeping its correctness [38, 39, 40], nonetheless they focus more on interface definitions (in terms of type and behaviour of ports and connectors) rather than on the issue of explicitly instantiating instance-to-instance links between components. Even component-based design tool implementations, as the solution proposed in [41], seem to miss out the problem we are addressing in this chapter.

Concerning the tools providing code generation from UML models, different solutions are provided when coming to the generation of interconnections between component instances. Enterprise Architect[2], by Sparx Systems, provides code generation from class diagrams where classes are linked through associations. More specifically, code is generated so that instances of the association's target class are owned by the instances of the source class. In our solution we target component-based design for encapsulation reasons and we

[2] `http://www.sparxsystems.com.au/`

aim at generating code which preserves this paradigm, that is to say components communicating by invoking functionalities on their own required ports with no need of knowing which component is on the other side of the connection providing the functionality. In this way generated code can be consistent to what specified at modelling level in terms of components and preservation of system properties from models to code is facilitated [42].

Other solutions, as in IBM Rational Rhapsody[3], maintain the generality of UML when coming to matching the multiplicities of components and ports. In fact, no decision is automated regarding the interconnections between components via ports, but rather instances generated according to their multiplicities along with function handlers (i.e., `get` and `set`) for managing connections when needed. In this case, the modeller will have to specify how to connect the different component instances when describing the behaviour of the single components.

Our approach provides an interpretation of the UML metamodel's semantics, as prescribed by the notion of semantic variation point, in order to automate the generation of fixed interconnections between component instances whose manual specification would require heavy and error-prone modelling effort in case of complex industrial systems composed by several hundred thousands components. Therefore, despite analysis, simulation, and code generation techniques remain valid without any form of components' instantiation automation, such a problem can remarkably affect scalability as well as error-proneness if not automated.

[3] `http://www-01.ibm.com/software/awdtools/\-rhapsody/`

Chapter 6

Generating Intermediate Concepts

The information about components and ports instances as well as the explicit links among them carried by InstanceM is employed together with the information represented by the state-machines in terms of states and transitions carried by ChessM to generate classes, attributes and functions in the intermediate representation (InterM). This represents the second step in the code generation process and is implemented as a QVTo M2M transformation taking as input the ChessM and the instances information stored in InstanceM; a first version of InterM is produced in output as description of the system's structure by means of intermediate concepts.

The main transformation rules are summarised by means of pseudo-code in Algorithm 2. The first part of the transformation is in charge of translating UML primitive types (pt2type(..)) and data types (d2class(..)) to the corresponding elements (i.e., primitive type and class) in InterM. Moreover, complex types such as enumerations and structures are translated too (d2class(..)). Auxiliary structures, as superclasses generally defining state-machine, state and message classes are created to be extended later on by specialising classes.

For each interface defined in ChessM a counterpart class in InterM is created and for each operation defined in the interface a mirror operation is defined and added to InterM (op2operation(..)). Moreover, for each interface's operation, the transformation generates an internal `send` operation in InterM for communication purposes (op2message(..)) that will be implemented in different ways depending on the targeted solution (see Chapter 8).

Finally, taking as input the component instances carried by InstanceM and the corresponding state-machine specifications in ChessM, the transformation generates the corresponding classes in InterM (ci_sm2class). Regarding the translation of state-machines, our approach resembles the *state design pattern*, as defined in [43], considering the component owning the state-machine as the *context* for the related states. Therefore each component instance is mapped to a class whose attributes represent the component's state-machine states (which in turn are translated into classes) and the properties defined in the component itself; component's operations are added to the generated class as operations (AddOperations()).

The final step consists of creating, for each of the state-machine's states, references to the outgoing transitions in order to be able to update the state of the state-machine when triggering transitions.

Algorithm 2 M2M Transformation from ChessM-InstanceM to InterM

```
Chess2Intermediate(in ChessM, in InstanceM, out InterM){
for each primitiveType pt in ChessM do
    pt2type(in ChessMM::PrimitiveType, out InterMM::PrimitiveType);
end for
for each dataType d in ChessM do
    d2class(in ChessMM::DataType, out InterMM::Class);
end for
for each interface i in ChessM do
    i2class(in ChessMM::Interface, out InterMM::Class);
    for each operation op in i do
        op2operation(in ChessMM::Operation, out InterMM::Operation);
        op2message(in ChessMM::Operation, out InterMM::Class);
    end for
end for
for each (component, stateMachine) ci_sm in InstanceM, ChessM do
    ci_sm2class(in ChessMM::StateMachine, in InstanceMM::Component, out
    InterMM::Class);
    AddOperations();
    for each state st in sm do
        st2class(in ChessMM::State out InterMM::Class);
        for each outTransition ot in s do
            ot2reference(in ChessMM :: Transition, out InterMM :: Reference-
            Access);
        end for
    end for
end for
}
```

6.1 Traceability and Back-propagation Model

The generation of intermediate concepts entails also the creation of traceability information between modelling elements (i.e., components, ports, operations and EFPs) and code units (in terms of intermediate concepts) which is stored in BackM to enable further back-propagation activities. BackM is created during the generation of InterM through a set of M2M transformation rules defined as part of the QVTo transformation presented above.

Navigating ChessM from the root component through all its composition levels, for each component instance a number of trace elements are created to keep track of the EFPs defined at both component and function level. The transformation rules in charge of creating the trace elements, related to each of the instantiated component instances operates according to the pseudo-code in Algorithm 3. More specifically, the algorithm is composed of three main steps:

- **Step 1 –** Starting from the root, for each component contained by the current component, an element of type `ModelElementInstance` in BackM is created. Particularly important in this process is that the containing component is set as *parent* of the `ModelElementInstance` in order to maintain the containment hierarchy crucial for back-propagation activities. In fact, it may happen that different instances of the same component type are defined in different parts of the model with ambiguous identities; by maintaining the containment relationships from the root component we are able to univocally identify the different instances of a same component type and correctly perform back-propagation. Moreover, since at code level the different component instances are identified by a progressive unique numerical identifier assigned during the code generation, the model element instance will also need to inherit this information in order to allow correct injection of the observed values to the right placeholder in BackM;

- **Step 2 –** For each operation defined in the component, a corresponding `FunctionalUnit` is created in BackM together with the elements representing the EFPs (`FunctionalUnitProperty`) that are modelled, and therefore could be monitored and back-propagated, as annotations for `FunctionalUnit`. An example of such could be `respT` and `memorySizeFootprint`, respectively representing execution time and memory allocation in the CHESS-ML ≪*CHRtSpecification*≫ stereotype derived from MARTE's ≪*RtSpecification*≫. The generation of properties at component level is performed in a similar manner, but `ModelElementProperty` elements are created instead of `FunctionalUnitProperty` and associated to `Model-`

`ElementInstance` elements instead of `FunctionalUnit`. Moreover, a corresponding C++ function is created and its execution will be used for monitoring and compute values for the defined properties. An `ExecutableUnit` element is created for this purpose, and the previously defined `FunctionalUnitProperty` elements are linked to it in order to complete the traceability chain (model-code-properties);

- **Step 3 –** Finally a `TraceElement` is created for each of the properties.

Algorithm 3 Algorithm for the Creation of Trace Links

```
Create_trace_links(in ChessM, out BackM){
for each comp in ChessM do
    ME = new BackMM::ModelElementInstance;
    for EFP_deco in comp do
        MEP = new BackMM::ModelElementProperty;
        TE = new BackMM::TraceElement;
        TE.modelSource = ME;
        TE.property = MEP;
        BackM.traceElements += TE;
    end for
    for each op in comp do
        FU = new BackMM::FunctionalUnit;
        EU = new BackMM::ExecutableUnit;
        for EFP_deco in op do
            FUP = new BackMM::FunctionaUnitProperty;
            FU.has += FUP;
            EU.monitors += FUP;
            TE = new BackMM::TraceElement;
            TE.modelSource = ME;
            TE.sourceFunction = op;
            TE.targetFunction = EU;
            TE.property = FUP;
            BackM.traceElements += TE;
        end for
    end for
end for
}
```

In Figure 6.1 the details of one of the trace elements created during the code generation process for the AAL2 subsystem is depicted. The meaning of this trace can be summarised as follows: the trace element *client2NodeConnect_2_NCC_ci_client2NodeConnect_respT* represents the trace link between the *client2nodeConnect* operation, and the property *respT*, defined for the component instance *NCC* and monitored through the code function *NCC_ci_client2nodeConnect*.

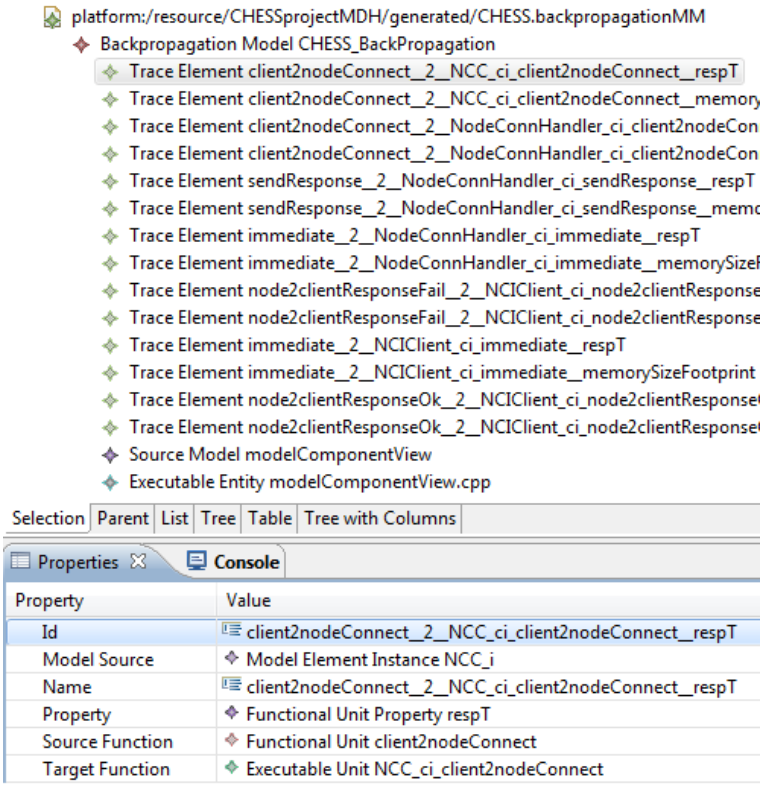


Figure 6.1: Trace Element details in the AAL2's Back-propagation Model

6.2 Summary and Related Work

In this chapter we presented the transformation step in charge of translating all the structural information contained in ChessM and InstanceM in terms of component instances, ports and links as well as state-machines to intermediate concepts (InterM). The transformation is defined through the QVTo transformation language and packaged as an Eclipse plugin as part of the generation chain from ChessM to C++. Moreover, explicit traceability links are generated and stored in an ad-hoc back-propagation model (BackM) that will be employed for back-propagation activities.

In the literature several attempts to the code generation from diverse modelling artefacts can be found. In [44] the authors propose a code generation solution to produce C tailored for real-time embedded systems from AADL focusing on flexibility of the code generator. This supports the reasoning about the multi-step approach proposed in our solution thought to be highly flexible and adaptable to different target platform languages.

The usefulness of introducing intermediate artefacts (i.e., InterMM in our solution) for mitigating the differences in expressiveness between modelling and target platform languages is confirmed by [26]. In our solution we prefer to place intermediate artefacts at the same abstraction level as the design models in order to maintain domain-independence and enhance reusability.

Several works, such as [45, 46, 47, 48, 49], just to mention a few, provide solutions similar to ours from an abstract perspective (i.e., using UML profiles and, except for [48], state-machine diagrams as source artefacts), though not focusing on generating full-fledged code exclusively from modelling artefacts.

Chapter 7

Completing Intermediate Model with Behavioural Descriptions in ALF

As prescribed in its specification [20], the execution semantics for ALF is specified by a formal mapping to foundational UML (fUML), which is a UML's subset defining a basic virtual machine for it, the abstractions supported by it, and thereby enabling conforming models to be translated into diverse executable forms for different purposes, such as verification, integration, and deployment [50]. There are three prescribed ways in which ALF execution semantics may be implemented [20], summarised as follows:

- **Interpretive Execution:** the ALF code is directly interpreted and executed;
- **Compilative Execution:** the ALF code is translated into a UML model conforming to the fUML subset and executed according to the semantics specified in the fUML specification;
- **Translational Execution:** the ALF code, as well as any surrounding UML concept in the model, is translated into some executable form on a non-UML target platform, and executes on it.

In this chapter, we present a solution towards the *translational execution* of ALF, using the UML–ALF implementation and related facilities (e.g., editors,

parsers, metamodels) provided along with Papyrus. This solution complements the transformation steps presented in the previous chapters enriching InterM with complex behavioural descriptions thus enabling the generation of full-fledged code.

Moreover, there are three levels of syntactical conformance defined for ALF, namely *minimum*, *full*, and *extended* [20]. In this work we focus on the *minimum* conformance and we provide translation of most of the entailed concepts (as depicted in Appendix C). The set of translated concepts, although limited if considering the expressiveness provided by ALF, reflect the ones which are usually found and used in the target language (and target domain).

Additionally, we delimited the number of state-machines to one per non-composite component and we defined ALF code at component operation level. Behaviour of state-machine transitions is defined within the component operation triggering the specific transition[1].

7.1 Transforming ALF to Intermediate Model

As described in the previous chapter, the structural specification of the system defined in ChessM is translated into intermediate concepts and stored in InterM. In order to achieve full-fledged code generation, we now need to complete it with the behavioural descriptions which are defined in ChessM in terms of ALF code within components' operations. An in-place M2M transformation defined using QVTo takes as input an ALF operation model (AlfOpM), translates its elements into their counterpart in the intermediate representation, and places them into the right placeholders in InterM.

Thanks to a dedicated parser provided by Papyrus under Eclipse, the action code related to each operation can be retrieved (Figure 3.1.d in Chapter 3) and manipulated as a model, which would be conforming to the ALF operation metamodel[2].

Once parsed into AlfOpM, the ALF code is translated to intermediate concepts through a transformation taking as input AlfOpM, ChessM (or more generally a UmlM) and InterM and providing an enriched version of InterM as output. For each of the parsed ALF operations present in UmlM, the related

[1] This is due to some limitations of the ALF editor provided in Papyrus that is still undergoing enhancements.

[2] The ALF operation metamodel is part of the ALF metamodel [20] and for simplicity reasons we consider the ALF operation models (AlfOpM) as conforming to the ALF metamodel (AlfMM) leaving apart the ALF operation metamodel

operation body is navigated and for each of the found ALF statements the appropriate handler function is called in order to translate it into intermediate concepts. Each of these handlers employs in turn further helpers and queries whose size varies from few to several hundreds of lines of code (e.g., 280 lines is the size of the transformation rule translating boolean expressions). An example partially representing the translation of the `if` statement is depicted in the next section.

The concepts in the minimum conformance that are currently left out from the translation process, most of which not commonly used in the target language (and domain), are: Behavior Invocation Expressions, Feature Invocation Expressions, Super Invocation Expressions, Link Operation Expressions, Class Extent Expressions, Sequence Operation Expressions, Sequence Reduction Expressions, Sequence Expansion Extensions, Isolation Expressions, Classification Expressions, Conditional-Test Expressions, Annotated Statements, Empty Statements, accept Statements, and classify Statements. It is important to notice that the translation of ALF concepts is independent of the underlying UML, and this makes the related transformation process reusable in other development processes which are based on UML profiles.

7.2 Applying the Solution

In order to show an example of input and output of the transformation process we consider the state-machine in Figure 4.2 of Chapter 4, which represents the behaviour of the AAL2's `NodeConn` component and for which we focus on the ALF code specifying the operation `NodeConn_riDisconnectCfm`. In this case the transformation process operates as follows.

First, an `Operation` is created with `Signature` set as *NodeConn_riDisconnectCfm* and owning passed parameters *portId* and *serverConn_r* defined as `ScopedNamedInstance`. Then, an IfStatement is encountered and handled. A control statement `If` is created as well as a `ConditionalBlock` and an `ElseBlock`. `ConditionalBlock` will contain a `Binary` expression as condition (representing the if condition) and two `Assign` statements. The `Binary` expression is of type *AND* (i.e., logic AND) and combines two furter `Binary` expressions of type *EQ* (i.e., equal to). Moreover, value accesses are used to represent the single variables within them. `ElseBlock` will contain the body of the else (empty for the two outer if statements in Figure 4.2). Within the IfStatement two more nested IfStatement are encountered and properly handled. The transformation rules translating an *IfStatement* are

depicted in Figure B.1 in Appendix B.

To grasp how value accesses, assignments and invocations work, let us consider the two `Assign` statements mentioned above. The first will be composed of a `ValueAccess`, in turn made of an `IndexAccessPart`, for representing "connHalf[serverConn_r.connHalf]"), and an `AccessPart`, for representing ".respondState", as left hand side, as well as a `FixedValue` set to *RI_RESPONDED_CFM* as right hand side. The second `Assign` contains, except for an `AccessPart` to a newly defined variable *secondHalf* (as `ScopedNamedInstance`) as left hand side, an `Invocation` of the operation with `Signature` *NodeConn_getSecondHalf* and a `ValueAccess` for the parameter "serverConn_r.connHalf".

7.3 Summary and Related Work

In this chapter we described an automatic mechanism for the translational execution of ALF, meant as the translation of the ALF text, as well as surrounding UML concepts (addressed in the previous chapters), into a non-UML target language to be executed on a non-UML target platform. The mechanism is defined through the QVTo transformation language and packaged as an Eclipse plugin which is employed in the generation chain from ChessM to C++. At the best of our knowledge, no documented attempt can be found in the literature concerning the definition and implementation of transformation mechanisms towards the translational execution of ALF, thus confirming the novelty of the contribution.

In the literature works such as [27, 28] generate code exploiting XML-based formalisms and scripts as well as specifying behaviours by means of target languages (e.g., Java) instead of model-aware formalisms such as ALF. In this way, consistency at modelling level may be jeopardised since the abstraction gap between modelling and programming languages does not permit native code from being aware of modelling concepts.

The generality of InterMM (and somewhat its intricacy) allowed us to be able to translate any combination of (the covered) ALF statements and expressions to intermediate concepts. Examples of this could be complex conditional logic expressions embedding multiple nested invocations, value accesses, and indexed values accesses, just to mention a few.

Chapter 8

Generating Full-fledged C++ from Intermediate Model

At this phase of the generation, InterM is complete and the final step of generating C++ code can be carried out. A M2T transformation defined by using the Xpand language is in charge of generating the actual C++ taking as input the sole InterM. Totally, the transformation is composed by the following five template files containing transformation rules:

- `Expressions`: definition of the transformation rules translating the intermediate concepts concerning expressions (i.e., `Expression` and specialising elements in Figure A.3) to C++;
- `Statements`: definition of the rules that take care of transforming the intermediate concepts concerning statements (i.e., `Statement` and specialising elements in Figure A.2) to C++;
- `Declarations`: definition of the transformation rules that transform the intermediate concepts (e.g., variables, methods, classes) into C++ forward declarations;
- `Implementations`: definition of the transformation rules that generate the implementation of the methods defined in InterM;
- `Main`: representation of the core template that, exploiting the other ones, generates a C++ header (.h) and a C++ implementation file (.cpp).

```
1 /** impl | If
2  * If statement sequence, consists of ConditionalBlocks
3  */
4 «DEFINE impl FOR If»
5  «EXPAND impl FOREACH ifblocks SEPARATOR "else"»«IF finalelse.size > 0»
6  «ENDIF»«EXPAND impl FOR elseBlock -»
7 «ENDDEFINE»
8
9 /** impl | ConditionalBlock
10  * One conditional block; if(condition){body}
11  */
12 «DEFINE impl FOR ConditionalBlock -»
13   if(«EXPAND template::Expressions::impl FOR condition»){
14       «EXPAND template::Declarations::member_decl FOREACH variables -»
15       «EXPAND impl FOREACH body -»
16   }
17 «ENDDEFINE»
18
19 /** impl | ElseBlock
20  * One else block; else{body}
21  */
22 «DEFINE impl FOR ElseBlock -»
23   else {
24       «EXPAND template::Declarations::member_decl FOREACH variables -»
25       «EXPAND impl FOREACH body -»
26   }
27 «ENDDEFINE»
```

Figure 8.1: Xpand rule for If-statement

Functional extensions have been defined in terms of the Xtend[1] language in order to lighten the verbosity of the transformation rules and increase their readability and understandability. Moreover, by exploiting the notion of polymorphic template invocation, we were able to considerably contain the size of the transformation both in number of rules and lines of code.
In Figure 8.1 we depict the Xpand rule that generates the structure of an *if* statement defined in the `Statements` template. Once such a statement is found in InterM, each of the *conditional blocks* (line 5), meant as the first *if* as well as any of the eventual following *else if*, is "expanded" by calling the appropriate rule (lines 12-17) and thereby the *if* skeleton is generated. Then, the *if* condition is built by a set of rules defined in the `Expressions` template, depending on which composition of expressions the conditional expression is composed of. The body of the *if* is generated eventually by (i) declaring variables in the scope of the *if* block (line 14) through appropriate rules defined

[1] `http://www.eclipse.org/modeling/m2t/?project=xpand`

in the `Declarations` template, and (ii) unwinding the body by calling the appropriate rule in the `Statements` template for each of the statements composing it (line 15). Finally the *else* statement is generated in a similar manner (lines 22-27). The transformation workflow defined in the `Main` template is summarised by means of pseudo-code in Algorithm 4.

Algorithm 4 M2T Transformation from InterM to C++

```
Intermediate2Cplusplus(in InterM, out [*.h, *.cpp]){
  createHfile(){
    generateDefines(in InterM.defines);
    generateTypeDefs(in InterM.types);
    generateGlobalVarsDeclaration(in InterM.globalvariables);
    generateInjectionMarkers(in InterM.markers);
    generateClassesDeclaration(in InterM.types)
    generateFunctionsDeclaration(in InterM.functions);
    generateDatatypeDeclaration(in InterM.types)
  }
  createCPPfile(){
    generateStructs(in InterM.structs);
    generateGlobalVars(in InterM.globalvariables);
    generateOSEprocesses(in InterM.processes);
    generateFunctionsImpl(in InterM.functions);
    generateOtherMethodsImpl(in InterM.types)
  }
}
```

The transformation takes as input InterM and produces in output header (.h) and implementation (.cpp) files, while configuration and `make` files are statically defined since they do not depend on the concepts carried by InterM. For generating the header file, InterM is navigated in order to translate definitions (i.e., defines, types, injection markers) and declarations (i.e., global variables, classes, functions, complex data types by means of classes). Afterwards, the implementation file is produced by navigating InterM in order to generate declarations (i.e., structures, global variables, OSE processes) and implementations of concepts declared in the header file (i.e., functions and other methods). A portion of the code generated (both header and implementation files) for the AAL2 subsystem is depicted in Appendix D in Figures D.1 and D.2, where the focus is on a subportion of the code concerning the `NodeConn` component instances including the code related to the ALF code specifying the behaviour of the operation `NodeConn_riDisconnectCfm` (modelled in Figure 4.2).

In respect to the M2M transformation process that manipulated ChessM to get InterM, which is closer to the type of language we aim to generate (i.e.,

C++), the M2T transformation task for the translation of modelling concepts from InterM into C++ is less intricate. This characteristic together with the generality of the concepts defined in InterMM makes it possible to implement other M2T transformations for generating code targeting different languages (e.g., Java or C#).

8.1 Deployment and Platform Configurations

The provided solution entails the generation of code for two different deployment configurations, namely single and multi process, and targeting Linux and OSE. On the one hand, when targeting Linux the application is generated as single process and, in our case, no particular deployment model is required. On the other hand, in case the generation targets OSE a detailed deployment model is needed for the transformation process to produce either single or multi process applications. The ability to handle parallelisation, transparently from the underneath hardware, is provided by OSE as described in Chapter 2.

For describing deployment information in CHESS-ML we exploit specific concepts by which the modeller defines allocation of component instances to processes through specific concepts provided by MARTE. Regarding specifically OSE, the deployment of component instances to the processing nodes is achieved through two intermediate layers: the OSE process and the OSE module. In fact, a component instance is allocated through a one-to-one connection to a specific OSE process; having a one-to-one connection allows monitoring at component level through the related process as described in the next Chapter. OSE processes are then allocated to OSE modules through a many-to-one connection; the module represents the running unit that will be deployed on the actual processing unit. In Figure 8.2 a portion of the deployment model concerning the AAL2 subsystem is depicted. More specifically the following allocations are shown:

- The two instances of component `NCIClient` are allocated to module `mod1` through, respectively, processes `mprocA` and `mprocB` (Figure 8.2.a);
- The single instance of `AAL2RIClient` is allocated to module `mod2` through process `mprocC` (Figure 8.2.b);
- The single instance of `NCC` is allocated to module `mod3` through process `mprocE` (Figure 8.2.c).

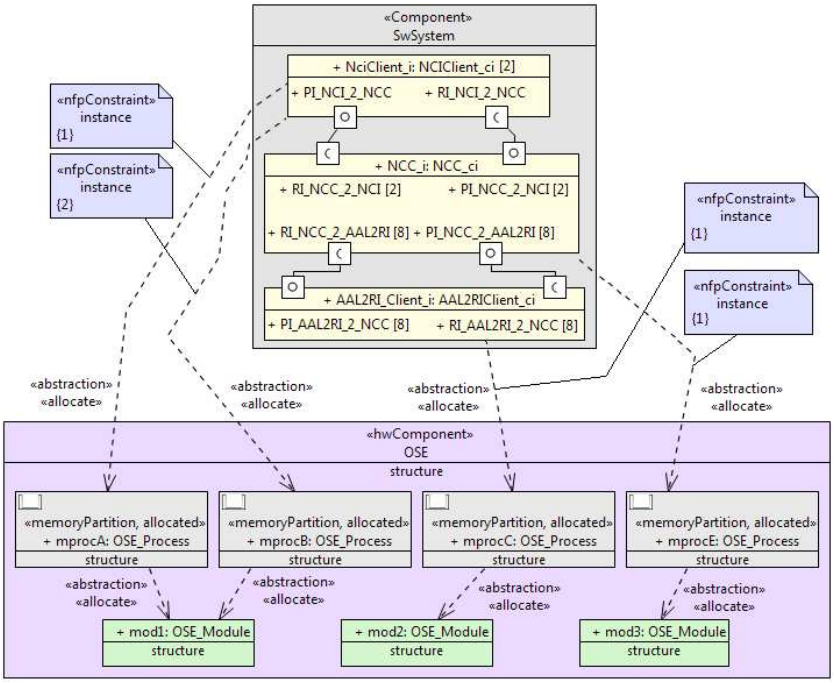


Figure 8.2: Partial Deployment Model of the AAL2 Subsystem

The OSE processes are defined as elements of type `OSE_Process` and stereotyped with MARTE's ≪*MemoryPartition*≫, while the OSE modules are typed as `OSE_Module`. The allocation is modelled by MARTE's ≪*allocated*≫ (on allocated components, process and modules) and ≪*allocate*≫ (dotted arrows between elements with allocation relationship). Moreover, the specific instance of the component which is meant to be allocated to a process is specified through a MARTE's ≪*nfpConstraint*≫ labelled `instance`.

In order to allow generation towards different platforms, the transformation process has to take into account the information carried by the deployment model that, together with the functional model, drives the code generation. More specifically, in the case of OSE applications, that information is exploited to create processes and modules as well as to statically deploy component instances on them. Moreover, the deployment configuration drives the generation of the communication code in order to distinguish between:

- **Intraprocess communication**: communication between component instances deployed on the same process, which is achieved by function calls;
- **Interprocess communication**: components deployed on different processes, which communicate via signals across processes.

Practically, the communication between components defined in ChessM in terms of ALF as function calls on ports had to be properly translated into appropriate intermediate concepts. Depending on the deployment configuration of the communicating components, a function call in the model can be translated into (i) a function call, in case of a single process application, or (ii) into a message `send` (i.e, OSE signal) in the case of multi process application. This is an example of application-specific decisions that had to be taken when implementing the code generation process. In any case, this does not prevent the code generator to be adapted to other communication paradigms depending on the specific domain and application under development.

8.2 Summary

In this chapter we described the last step of the code generation process, namely the transformation from the intermediate model (InterM) to the target language (C++). The transformation is defined through the Xpand language and it is part of a set of Eclipse plugins providing the generation chain from ChessM to C++.

Additionally we depicted how to model deployment information in CHESS-ML in terms of the allocation of components instances to processes and modules through specific concepts provided by MARTE. The deployment model is crucial for targeting different platforms (Linux and OSE). In fact, different deployment configurations affect the M2T transformation since communication patterns among components vary depending on them.

In the scope of the CHESS project, the ability to produce OSE tailored code, with its specific libraries and APIs, enabled the possibility to validate the code generation process against industrial case-studies in the telecommunication applicative domain. Moreover, the monitoring capabilities built upon OSE permitted to employ the round-trip approach for deployment assessment, as described in the next chapter.

Chapter 9

Code Execution Monitoring and Back-propagation

At this point of the development using our round-trip approach, the target code has been automatically generated and therefore the forward path is completed. The next operation is monitoring the code execution on the specific platform to gather runtime values for the selected EFPs, and then back-propagate them to modelling level.

For the provision of back-propagation capabilities the approach had to overcome common reverse engineering challenges in mapping data derived from data analysis (e.g., monitoring results) to more abstract design levels [51]. This is usually achieved by supporting iteration of the process and bidirectional mappings from models to data analysis and vice versa [51]. Our solution achieves back-propagation through a set of model transformations (T2M and M2M, both in-place) which enriches the design model with the observed values gathered at system implementation level by monitoring activities.

The back-propagation process is composed of:

- *Monitoring results and traceability information management*: results coming from the monitored execution of the generated code are part of the source artefacts for back-propagation to the design models; the representation format of this information is pivotal. Monitoring results are manipulated in order to extract the observed values and store them in formal structures to be fed to the back-propagating transformations. The proposed solution provides storing structures as part of the back-

propagation metamodel (BackMM). Observed values as source for the back-propagating transformations are not enough. In fact, the traceability chain defined along the path from design model to observed values is also part of the source artefacts to be fed to the transformations in order to correctly propagate values back to the design model. Moreover, regarding the code inserts needed for monitoring activities, they are automatically generated with the rest of the implementation, thus not jeopardising the consistency between source model and code;

- *Annotation of design models*: the final step of the round-trip approach is the enrichment of the design model with values gathered at system implementation level. The enrichment is performed by injecting the observed values into the related model elements' placeholders at design level.

Once completed, the back-propagation task produces an extra-functionally decorated version of the initial design model. At this point it is possible for the developers to evaluate it and possibly operate modifications and optimisations when needed. The process might necessitate multiple iterations in order to reach the desired quality level, in terms of EFPs, required by the system specification.

Depending on target platform and available monitoring features, selected EFPs can be monitored at a specific level of granularity and back-propagated to the design model for, e.g., comparing expected with observed values. More specifically, regarding applications generated targeting Linux we were able to exploit specific APIs to monitor total execution time and allocated memory at (component's) function level. Concerning multi process applications for OSE, we could gather a wider set of EFPs both at component and system level.

The way to perform the injection of monitoring results to BackM depends on both code generation and output format of the monitoring activities; it could in fact vary from M2M to T2M transformation (or the combination of both). This can be considered a variable point of the round-trip approach in the sense that it is hard to generalise for a multitude of different tools. In this work we implement this injection by means of an in-place T2M transformation since the monitoring activities (both in Linux and OSE) give a textual description of the computations as output.

In the next sections we describe the mechanisms, in terms of model transformations and involved intermediate artefacts, for code execution monitoring and back-propagation for both function level monitoring in Linux and component level monitoring in OSE.

9.1 Monitoring and Back-propagation at Function Level in Linux

Once C++ for Linux is generated from the AAL2's ChessM, we employ the API *getrusage* [52] for monitoring its execution at function level. The monitoring's outcome is a log file made of a set of four-token lines formatted as: `ExecutableUnit ModelElementInstance.id Property Value`. In Listing 9.1, an example of the log file specific to our example is depicted. In the first row we can see that a response time of 1201 milliseconds has been observed for the function `NCIClient_ci_node2clientResponseFail` in the component instance with $ID = 13$.

Listing 9.1: Monitoring Log File

```
NCIClient_ci_node2clientResponseFail 13 respT 1201
NCIClient_ci_node2clientResponseFail 13 memorySizeFootprint 8716
NCC_ci_nodeConnResp 2 respT 3402
NCC_ci_nodeConnResp 2 memorySizeFootprint 12327
NodeConnHandler_ci_client2nodeConnect 12 respT 6004
NodeConnHandler_ci_client2nodeConnect 12 memorySizeFootprint 4550
NetConn_ci_netConnRequest 3 respT 1457
NetConn_ci_netConnRequest 3 memorySizeFootprint 6093
PortHandler_ci_nodeClientRequest 7 respT 3990
PortHandler_ci_nodeClientRequest 7 memorySizeFootprint 8770
AAL2RIClient_ci_connNodeReq 14 respT 1805
AAL2RIClient_ci_connNodeReq 14 memorySizeFootprint 9982
```

The injecting in-place T2M transformation is implemented in Java and, taking as input the BackM and the monitoring log file (LinuxMF), acts according to Algorithm 5.

Algorithm 5 Log to BackM Injection Algorithm

for *each line* l *in* $LinuxMF$ **do**
  $execUnit = l[1]$;
  $id = l[2]$;
  $property = l[3]$;
  $value = l[4]$;
  $BackMtrace = BackM.search(execUnit, id, property)$;
  if $BackMtrace! = NULL$ **then**
    $BackMtrace.property.value = value$;
  end if
end for

LinuxMF is navigated and each line is tokenised according to the defined format; the tokens `ExecutableUnit`, `ModelElementInstance.id` and `Property` represent the information for which a match has to be sought in BackM. The identifier (represented by `ModelElementInstance.id`) related to the `ModelElementInstance` is crucial for identifying from which component instance in the code the observed value is derived and thereby to which trace element has to be injected in BackM. Once a match is found, which is to say that there is a trace element `BackMtrace` linking `Property` with `ExecutableUnit` and `ModelElementInstance`, then the token `Value` is injected into the correct placeholder pointed by `Property`.

The resulting complete BackM for the AAL2 subsystem is shown in Figure 9.1.

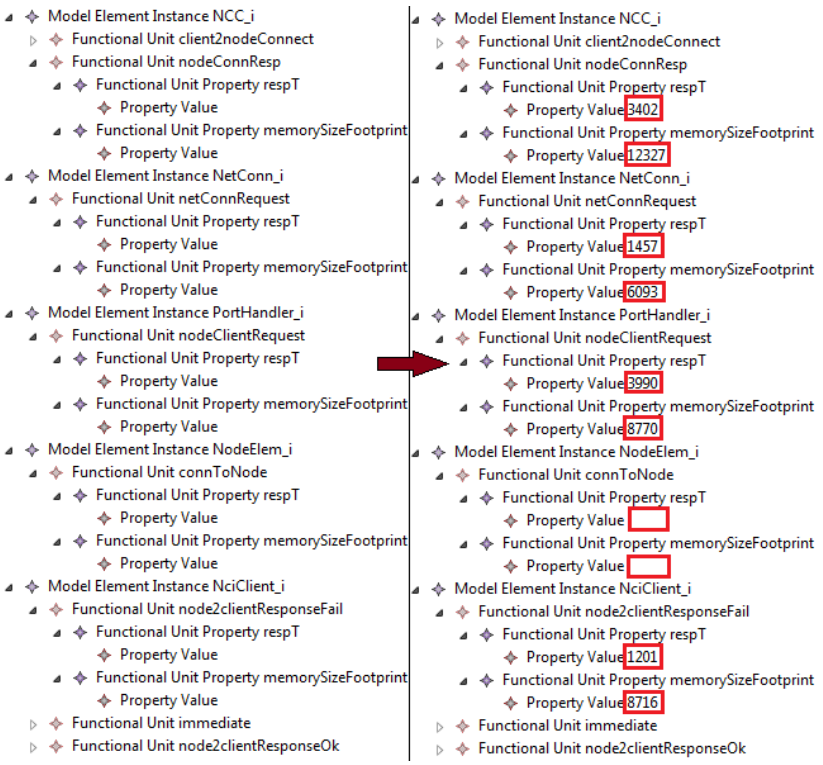


Figure 9.1: Injection of the monitoring results to the back-propagation model

Note that, since during monitoring activities no values were gathered for the properties defined for the functions of the *NodeElem* component, no value is injected into BackM. At this point all the information to drive the back-propagation to ChessM is stored in BackM. The injection of values is performed through a QVTo in-place M2M transformation. Taking as input ChessM and BackM, the transformation performs a set of in-place modifications on ChessM to enrich it with the monitored values stored in the BackM.

As defined in Chapter 2, BackM is composed by a non-empty set of trace elements *TE* defined as quadruples <*ME, FU, EU, FUP*> where *ME* is a model element contained in a design model *SM*. *FU* represents an operation/method defined in *ME* and *FUP* represents an EFP defined for *FU* and observed by monitoring the execution of the executable unit *EU*. The transformation algorithm (Algorithm 6) takes in input BackM and ChessM, as well as the metamodels to which they conform to; the output will be an enriched version of ChessM.

Algorithm 6 BackM to ChessM Injection Algorithm

for $each\ trace\ element\ TE = (ME, FUP)\ in\ BackM$ **do**
  $ChessMproperty = ChessMproperty.search(ME, FUP)$;
  if $ChessMproperty\ exists$ **then**
    $SMproperty.value = FUP.value$;
  end if
end for

BackM is navigated and for each trace element *TE* a match is sought in ChessM; if model element *ME* and property *FUP* traced by *TE* match with a corresponding pair in ChessM then the value associated to *FUP* in *TE* is injected into the matching property in ChessM.

In Figure 9.2 the AAL2 subsystem and its *NCC* composite component with back-propagated values for execution time and allocated memory are shown. More specifically the values are back-propagated to the apposite annotation stereotyped as ≪*CHRtSpecification*≫, which we defined in ChessMM as a modified version of MARTE's ≪*RtSpecification*≫. In this case, since the monitoring is performed at function level, the annotation is connected to the provided port of the involved component, which is the place where the function is exposed. Since the same port can be typed to different interfaces and expose several functions, through the property `partWithPort` we identify the specific interface and, within it, through `context` we specify which of the interface's functions the annotation is related to.

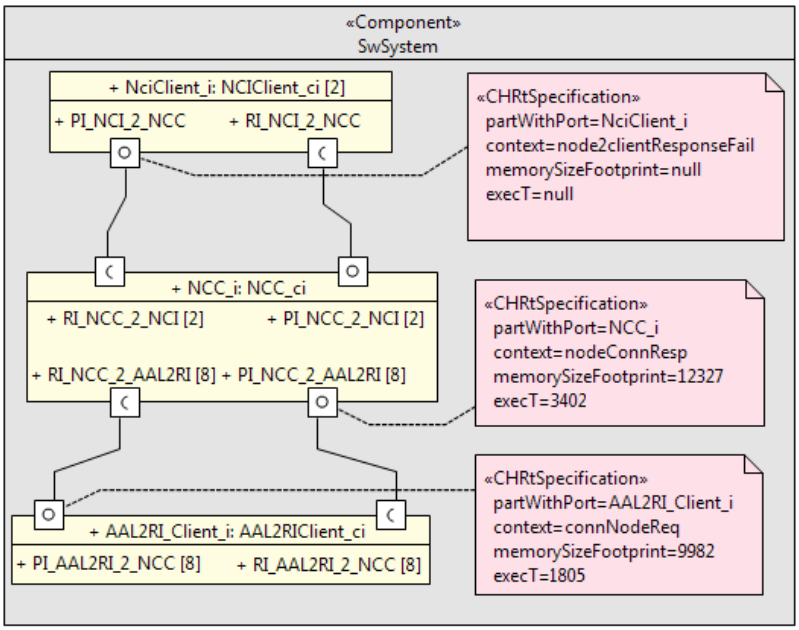


Figure 9.2: Back-propagation to AAL2 subsystem in CHESS

The back-propagated values are put in: `execT`, for execution time, and `memorySizeFootprint`, for allocated memory. The round-trip process has produced an extra-functionally decorated version of the design model and it is now possible for the developing team to evaluate monitored EFPs. The decorated ChessM could be employed, e.g., to perform analysis related to execution time, such as Measurement-based Worst-Case Execution Time (WCET) [53] which exploits the combination of model-based and code execution analysis for improving WCET analysis in embedded real-time systems. Eventually, optimisation activities can be performed directly at modelling level rather than at code level with a consequent conservation in terms of consistency among the artefacts and their properties. The process is meant to be iterated at will until the developer is satisfied with the resulting implementation.

9.2 Monitoring and Back-propagation at Component Level in OSE

In order to enable component level[1] monitoring in OSE, specific extensions to the execution platform have been performed and presented in [54]. These extensions have been implemented mainly in the form of two additional system processes: one for monitoring and another for logging. These two processes are assigned lower priorities than the generated application ones. The monitoring process is responsible for calculating and determining the values for EFPs of interest for both the whole system as well as per component. The actual task of logging this information is separated from the monitoring process and performed by the logging one. This separation allows to mitigate the side effects of resource-demanding I/O activities. When a request for monitoring is issued by one of the application processes, the monitoring process starts executing and determining EFPs' values. The information to be logged is sent to the logging process by the monitoring one through apposite signals. Therefore, if the logging process does not get the needed CPU time to perform its job, the signals sent to it are pushed in its signal queue, maintained automatically by OSE, and processed as soon as it gets to execute.

The implemented monitoring process is capable of determining values for the following properties:

- **System level properties:** total CPU load, total number of generated signals in the system, system throughput (sent and received packets), total number of processes in the system;
- **Component level properties:** total execution time of a component instance (from the startup of the system including all invocations of it), execution time (one invocation), response time, heap and stack usage, number of signals generated, and CPU load.

In our example we are interested in the component level properties and, among them, we take into account: total execution time of a component instance, response time, heap and stack usage since they can be represented through defined stereotypes at modelling level and therefore eventually used for model-based analysis based on MARTE.

[1] From this point on, we employ component level and process level monitoring as synonyms since, having a one-to-one relation, the values monitored for a specific process represent the values related to the component instance allocated to it.

To calculate execution and response times, `swap_in` and `swap_out` handlers of OSE have been used. The former event handler is invoked each time a process gets CPU to execute, and the latter is invoked when CPU is taken from a process and it is preempted. The algorithms and mechanisms for the calculation of execution and response times have been implemented into these two event handlers. However, since they are invoked for every process in the system, additional tweaks were made in order to filter their executions for only the generated application processes which are of interest.

The monitoring activities give a textual description of the gathered values as output. The results of monitoring the execution of the C++ code generated from the AAL2's ChessM and with the deployment configuration in Figure 8.2 are shown partially in Listing 9.2.

Listing 9.2: Monitored Properties

```
466,PROCID, mprocA,1003c,65596
476,S_CPU_LOAD, 29.9780
476,S_NUMBER_OF_PROCESSES, 73476
476,S_NUMBER_OF_SIGNALS, 365
479,S_THROUGHPUT, 942,1676
479,P_HEAP_USAGE, 384,512
479,P_STACK_USAGE, 1536,0,2048
479,P_NUMBER_OF_SIGNALS, 2
579,P_EXECUTION_TIME, 14
579,P_RESPONSE_TIME, 1609984
-----------------------------------
579,PROCID, mprocE,10040,65600
...
-----------------------------------
612,PROCID, mprocC,1003e,65598
622,S_CPU_LOAD, 9.8427
622,S_NUMBER_OF_PROCESSES, 73
622,S_NUMBER_OF_SIGNALS, 675
622,S_THROUGHPUT, 820,1676
623,P_HEAP_USAGE, 384,512
623,P_STACK_USAGE, 1536,0,2048
623,P_NUMBER_OF_SIGNALS, 2
623,P_EXECUTION_TIME, 11
623,P_RESPONSE_TIME, 1619979
```

The first column in Listing 9.2 indicates the time instance at which the monitoring has been performed (in system ticks unit). The second column identifies the type of the monitored information; the properties beginning with 'S_' indicate a system level value while the ones starting with 'P_' identify a process level value (e.g., S_NUMBER_OF_SIGNALS: total number of signals in the

system at the moment of monitoring, P_NUMBER_OF_SIGNALS: total number of signals owned by a process). The values after the name of the process (i.e., mprocA) indicates its ID (system unique identifier) in hexadecimal and decimal format respectively. As it can be seen, some of the properties have multiple values, in which case they mean different aspects related to the same property. For instance, the first value related to P_HEAP_USAGE represents the heap size requested by the process and the second one shows the actual heap size allocated for the process by the operating system (the difference between the two is due to factors such as memory paging and memory management mechanisms of OSE).

The results of the back-propagation are shown by means of extra-functional decorations of ChessM in Figure 9.3, where we can notice that values concerning *mprocA* are back-propagated to instance 1 of `NciClient` while the ones carried by *mprocC* apply to the single instance of `AAL2RI_Client`. This correspondence is stored in BackM and originates from the deployment model depicted in Figure 8.2. The back-propagation process is, even in this case, a two-step chain consisting of an in-place T2M transformation from monitoring log file to BackM and an in-place M2M transformation from BackM to ChessM. The working principle is the same as the transformations presented in the previous section, with practical differences in the tokenization of the log file as well as the target elements for back-propagation in ChessM (component instance deployed on a specific process instead of function of a specific component). Once the monitored results have been back-propagated, the developer has at her disposal the modelled system enriched with actual values gathered at runtime. The values depicted in Listing 9.2 and propagated back to ChessM (in Figure 9.3) are related to the deployment configuration in which the component instance `NciClient[1]` is deployed to `mod1` through `mprocA` and the component instance `AAL2RI_Client[1]` is allocated to `mod3` via `mprocC`.

As for the function level monitoring, the values are back-propagated to the apposite annotation stereotyped as ≪*CHRtSpecification*≫. In this case, since the monitoring is performed at component level, the annotation is connected directly to the component; through the property `instance` we identify the specific instance the annotation is related to. The back-propagated values are put in: `heapSize`, for heap usage, `stackSize`, for stack usage, `respT`, for response time, and `execT`, for execution time. At this point, let us try out a different deployment configuration in which we allocate both `NciClient[1]` and `AAL2RI_Client[1]` to `mod1` since, e.g., the communication between them is quite dense. Once the model is modified, the code can be regener-

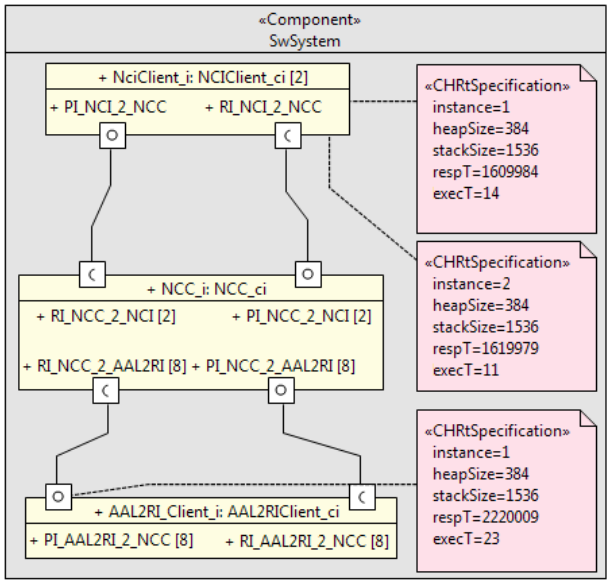


Figure 9.3: Decorated AAL2 model in CHESS

ated and its execution monitored. In Table 9.1 the monitoring results concerning both the first as well as the tuned deployment configurations are depicted. Configuration `1` represents the deployment of the two component instances on separate modules, while `2` concerns the deployment of both instances on a single module.

Configuration	Component Instance	Process	Module	Execution Time
1	NciClient[1]	mprocA	mod1	14
1	AAL2RI_Client	mprocC	mod3	**11**
2	NciClient[1]	mprocA	mod1	14
2	AAL2RI_Client	mprocC	mod1	**3**

Table 9.1: Different deployments that induce different monitoring results

As we can see, by changing the deployment configuration, in the specific case by allocating the two instances on the same module, we actually experience a decrease of the execution time of `AAL2RI_Client` from 11 to 3. Besides this reduction which may not be relevant in the actual employment of the sys-

tem, what we aimed at pointing out was the usefulness of having an automatic mechanism for gathering and back-propagating runtime values to model level for allowing thorough evaluation of the system's deployment configuration. While providing meaningful values at model level, the approach is not yet able to provide hints on how to interpret them. Limitations of the current solution in this sense as well a future enhancements towards further automation in the tuning phase are discussed in the Chapter 11.

9.3 Summary and Related Work

In this chapter we described the details of our novel step of back-propagation from runtime to modelling level for evaluating the preservation of selected system properties. Starting from monitoring the execution of the generated code on the specific platform, we showed how the values are gathered and the transformation means that have been designed and implemented to allow the annotation of design models with those values. Depending on the target platform's monitoring capabilities, we provided monitoring and back-propagation at two different levels of granularity, namely function level in case of Linux applications and component level in case of multi process OSE applications. Moreover, a possible employment of component level monitoring and back-propagation in deployment assessment has been depicted.

In the next paragraphs, our solution in terms of back-propagation, monitoring and employment of system implementation values for deployment issues is related to the state-of-the-art.

Back-propagation. Navabi et al. [55] in the early 90's, and some years later Mahadevan and Armstrong [56], came up with different approaches for back-annotating behavioural descriptions with timing information; however, both operate horizontally[2] in terms of abstraction levels and no automation is provided. It is worth noting that having a mechanism to automatically annotate the design model with monitored values is of critical importance in order to assist the developer in understanding at a glance the relationships between expected behaviour and model entities at design level, without having to inspect the related generated code.

[2]Horizontal and vertical are used for specifying the direction of data transitions among artefacts either at the same (i.e., horizontal, from model to model) or at different (i.e., vertical, from code to model) level of abstraction.

In the literature, Varró et al. propose in [57] back-propagation for enabling execution traces retrieved by model checkers or simulation tools to be integrated and replayed in modelling frameworks; even though some similarities to our approach might be found when dealing with traceability issues, the two approaches aim at solving two different problems. The most similar approach to ours is described by Guerra et al. in [58] where back-propagation of analysis results to the original model by means of triple graphical patterns is described. Nevertheless, the approach is meant to horizontally operate at modelling level with propagation of data among models. While, dealing with embedded systems, our approach focuses on vertically propagating analysis results observed at code level back to design models for better understanding of those EFPs that cannot be accurately predicted at higher levels of abstraction.

Monitoring. Regarding measurements of EFPs at system implementation level, besides runtime monitoring, other verification techniques (e.g., static analysis) can be used for small and simple systems, but their application for large and complex systems might not always be practical and economical [59]. Even in cases where such techniques are feasible, conditions that cause invalidation of the analysis results at runtime may happen. An example of such is the difference between the ideal execution environment (considered for performing analysis) and the actual one which leads to the violation of the assumptions that were taken into account when performing static analysis [6]. Therefore, the information gathered through monitoring the execution of a system is not only interesting and useful for observing the actual behaviour and to detect violations at runtime, but also to be used for making adaptation decisions, as well as to induct enforcement and preservation of properties.

Saadatmand et al.'s work in [60] serves as an example of using monitoring information for balancing timing and security properties in embedded real-time systems. In [61], they provide an approach for improved enforcement and preservation of timing properties in embedded real-time systems. Huselius and Andersson in [62] present a method for the synthesis of models of embedded real-time systems from the monitoring information collected from their execution. In this thesis, however, we exploit monitoring results, from which observed values are extracted and used to refine design models with EFPs' values detected during the execution of the system.

Deployment Assessment. In the literature there exist approaches dealing with deployment optimisation based on measurements at system implementation level as described in [63]. A dated approach by Yacoub in [64] introduces

systematic measurements of system properties for component-based systems, but no tool support is provided.

The COMPAS framework by Mos et al. [65] is a performance monitoring approach for J2EE systems. Components are Enterprise Java Beans (EJBs) and the approach consists of monitoring, modelling, and prediction. An EJB application is augmented with proxy components for each EJB, which send timestamps for EJB life-cycle events to a central dispatcher. Performance measurements are then visualised with a proof-of-concept graphical tool and a modelling technique is used for generating UML models with SPT annotations from the measured performance indices. Then, for performance prediction of the modelled scenarios, the approach suggests using existing simulation techniques, which are not part of the approach.

Based on the COMPAS framework, two further approaches have been proposed: AQUA, by Diaconescu et al. [66], and PAD, by Parsons et al. [67]. Both approaches expect working J2EE applications as input. AQUA focuses on adapting a component-based application at runtime if performance problems occur. The main idea is that a software component (EJB) with performance problems is replaced with one which is functionally equivalent from a set of redundant components. Furthermore, the approach involves monitoring the workload of a running application. PAD focuses instead on automatic detection of performance anti-patterns in running component-based systems. The approach targets EJB systems and includes performance monitoring, reconstruction of a design model, and anti-pattern detection.

The TestEJB framework by Meyerhöfer et al. [68] implements a profiling technique, which is application independent and more lightweight than available commercial J2EE profiling tools. Besides the execution times of individual EJBs, the framework is able to determine call traces from single users. The approach is based on the interceptor patterns and implemented as an extension to the JBoss application server.

The goal of these approaches is to identify performance problems in the running system and adapt the implementation (at code level) to make it able to fulfil EFPs requirements. Instead, the uniqueness of our round-trip approach consists in introducing a new dimension to deployment assessment at model level with the help of measurements gathered at system implementation level. In fact, when measurements are completed, the code is not manually tuned, but changes to the system are rather performed at model level from where code is re-generated. Doing so, consistency between models and code is kept and thereby the validity of decisions made at model level is likely to be preserved at code level (and the other way around). Moreover, by exploiting the accuracy of

system implementation level measurements at modelling level, the developer is relieved from complex code inspection and error-prone manual tuning of code.

Chapter 10

Validation

The round-trip approach proposed in this thesis has been validated both in-house and in industrial settings [69] at Ericsson Nikola Tesla (Zagreb, Croatia) under the leadership of Ericsson AB (Stockholm, Sweden). The generated code for the complete AAL2 system was compiled and executed on the actual target platform.

Concerning scalability, we analysed the behaviour of the approach from the perspective of the entire process as well as stepwise. Moreover, within the same case-study we tested several model sizes, as depicted in Table 10.1, on a machine running a 2.6GHz CPU and 8GB RAM in order to evaluate the transformations performance.

# Component instances	# Port instances	# Generated lines	Generation time
14	66	4270	52 sec
253	1600	6498	113 sec
2003	14000	20216	831 sec

Table 10.1: Details about size of models and related generation results

From the numbers reported in the table, one can notice that the increment in the number of generated lines is not proportional for the three presented cases. This is due to the fact that, since we consider *instances* rather than types, declarations in the header and implementations are only defined once per set of instances (at classifier level), while forward declarations as well as instantiation and initialisation are generated for each of the instances (at instance level). Given n as the greatest number of instances per component, m as the greatest

number of instances per port and k as the number of hierarchical composition levels, the general limit behaviour of the computation is represented by $O((n * m)^k)$; this is due to the fact that, in order to generate explicit instances and links, the model is navigated up to $(n * m)$ times to perform one step down from a higher hierarchical level to a lower one (for a total of k levels). Overall, from a process-wise perspective, the proposed solution on a model with $k = 2$, resulted very scalable up to $n + m = 10^3$ (i.e., within 5 minutes) while degrading going toward $n + m = 10^4$ (i.e., over 30 minutes); in any case the process always accomplished its goals.

Analysing this result from a stepwise perspective, we noticed, on the one hand, that the least scalable tasks were those responsible for the code generation. The reason stems from the computational complexity of the involved transformations. On the other hand, regarding the time needed for monitoring activities, it heavily depends on the duration of the code execution since the measurements themselves are mostly performed by parallel processes during the execution. Nevertheless, the calculation of more complex EFPs could introduce additional complexity hence requiring additional computation time.

Concerning back-propagation tasks, they resulted to be very scalable thanks to the detailed information, concerning the path to the specific model element to be annotated, carried by the back-propagation model. This means that most of the needed computation is carried out when generating the back-propagation model, while the actual values injection, first from monitoring results to back-propagation model and therefore from the latter to the design model only involves an update of specific values with no need of complex searches nor navigations.

The better scalability of back-propagation tasks (e.g., monitoring results management and actual back-propagation to design model) resulted to be less dependent on the design model's size. Intermediate artefacts' size may grow proportionally to design model's; the fact that they are meant to be transparent and handled only by the process itself relieves the developer of the burden of understanding and managing them and lowers possible overheads deriving from their graphical rendering.

Generally, the number of iterations for reaching the desired EFPs depends on the accuracy of measurements as well as the modeller's ability in both modelling the system and also effectively employ the back-propagated values to tune the models accordingly. That is to say that the developer is supposed to be able to understand the back-propagated values in relation to the expected behaviour and thereby tune the models accordingly to generate a better-performing implementation. In aid to the modeller, model-based analysis and

deployment optimisation techniques could be exploited to minimise the number of iterations.

On the Model Transformations The round-trip approach consists of a set of model transformations. In Table 10.2 we depict them with details regarding number of transformation rules as well as number of lines. Having the solution divided into smaller and separated transformation steps provides a number of advantages:

- **Debugability:** testing a transformation process of this size is not a trivial task. Having it divided into separated steps that are sequentially executed helps in debugging and verifying the transformation process. For instance, the single transformations have been tested[1] one by one (and within them rule by rule) focusing on coverage and determinism of the correspondence source-target;
- **Adaptability:** changes to the source modelling language (e.g., CHESS-ML) as well as the action language (e.g., ALF) may lead to the necessity of adapting the transformation process to account them. Since separated transformations have been defined for the translation of structural and behavioural concepts, changes to the one would not affect the other and vice versa, making the adaptation task less intricate;
- **Reusability:** the single transformation steps are independent from each other making them eligible, even individually, for reuse in other developments and tools.

Task	Transformation	Language	Type	# Lines	# Rules
Code Generation	ChessM to InstanceM	QVTo	M2M	419	13
Code Generation	ChessM to InterM	QVTo	M2M	1257	43
Code Generation	AlfM to InterM	QVTo	In-place M2M	3296	104
Code Generation	InterM to C++	Xpand	M2T	823	82
Back-propagation	Log to BackM	Java	In-place T2M	589	—
Back-propagation	BackM to ChessM	QVTo	In-place M2M	125	10

Table 10.2: Details about involved model transformations

Moreover, the exploitation of intermediate modelling artefacts (i.e., intermediate metamodel) gives us the possibility to easily extend the approach to enable

[1]Note that the transformation testing phase was mostly manual due to the scarcity of reliable automatic testing mechanisms for such intricate multi-source and multi-target transformations.

the generation of code in programming languages different from C++. This can be achieved by acting only on the M2T transformation, which represents the least intricate transformation step in the generation chain. Generation of C++ sister languages such as Java or C# would require lightweight adaptations to the existing M2T transformation while, for generating other types of languages, the M2T transformation might need to be reimplemented from scratch.

The translation of ALF to C++ (via intermediate model) was not trivial due to several reasons, ranging from different semantics of similar concepts to gaps in expressive power as well as differences in the allowed programming patterns the two languages comprise. In order to achieve a deterministic transformation able to generate C++ from a limited set (within the minimum conformance) of ALF constructs, we had to identify and find a solution for the languages' misalignment. In some cases, assumptions made in CHESS helped us out in this task.

An example of this could be, e.g., the way the two languages manage memory and objects. ALF, in many ways similar to Java, instantiates objects explicitly with the `new` expression. Declaring a variable with the type of a class or interface only declares a reference to an object. Moreover, in ALF the object instance is destroyed whenever there are no more references to it, and memory released by a garbage collector. C++ does not provide any automatic memory management; memory is in fact manually managed through `new` and `delete` expressions or the C constructs `malloc` and `free`. This semantic gap can be handled in many different ways; one solution could be to use the smart pointer design pattern or an intrusive reference counting pointer. Anyhow, since one of the core assumptions made in CHESS for ensuring preservation of certain EFPs from models to code is that dynamic instantiation of objects is forbidden, the code generation process could be relieved from issues regarding memory management. In fact, this assumption prevents the user from modelling memory allocations in terms of ALF constructs. Consequently, memory management (i.e., allocation and release) are only statically defined at transformation level depending on the target programming language when generating interprocess communication in terms of message passing.

Chapter 11

Discussion

In this section we present a discussion regarding the research challenges in relation to the solutions we propose for tackling them. Moreover we elaborate on general aspects related to the techniques and technologies that have been considered and exploited to achieve the presented solution.

11.1 Research Challenges and Solutions

Research challenge 1: ***"Define an automated process to enable the generation of full-fledged code from design models".***
The round-trip approach is meant to operate on models defined through UML and its profiles, hence the necessity of handling the ambiguity of UML soon arose. More specifically, since we employed the component-based design pattern defined in UML, we had to deal with the fact that, while the number of instances of components and ports can be precisely specified, the port-to-port links are not equipped with a detailed specification of the component instances they connect.

That is the reason why we developed means to automatically generate the set of actual links between explicit component instances by (i) defining semantic rules for fixing the UML's semantic variation point concerning this issue, (ii) identifying and developing appropriate means for storing the generated information (i.e., instance metamodel), (iii) and developing the actual transformation rules that would follow the defined semantic rules to generate instances and links. Concerning the semantic rules, since any combination of multiplic-

ities of components and ports is allowed in UML we had to circumscribe the problem and define a solution within our scope.

In this activity, we took advantage of one of the constraints given when defining the CHESS-ML according to which, for ensuring guarantees at runtime of the properties modelled at design time, dynamic instantiation of components is not allowed. Therefore in our solution we entail only prefixed cardinalities whose values are defined as concise (i.e., `[n]`) while range values (e.g., `[n..m]`) are left as future enhancement. In addition, the connectors linking components via ports have fixed multiplicity (i.e., `[1]`) thus leaving components' and ports' multiplicity as variables for the interconnections calculation and generation, and only binary connections are considered, leaving n-ary possibilities as future work.

The code generator presented in this work targets single process application for Linux and multi process applications for OSE. Future work are planned be directed to the enhancement of the approach for enabling multicore optimized code generation.

Research challenge 2: ***"Define and implement translational execution of ALF towards non-UML platforms".***
The employment of action languages for specifying complex behaviours within design languages is not new. Many different approaches can be found in the literature as described in Section 7.3. Nevertheless, there is an issue, often underestimated or even ignored, with many of those approaches which employ programming languages (e.g., Java or C) as action language: consistency at modelling level is very hard to keep since the abstraction gap between modelling and programming languages does not permit native code to be aware of modelling concepts. For this reason we decided to exploit a modelling formalism, namely ALF, for the definition of complex behaviours within design models. This decision, while preferable due to the aforementioned reasons, came at a cost. In fact, in the case of using programming languages for behaviour descriptions, no complex transformation is needed for translating it to the target language since they coincide. While in our solution, since ALF is a modelling formalism, a proper set of model transformation rules were needed in order to translate its concepts to our target language. Moreover, no documented attempt at translating ALF to target languages can be found in the literature. We provided a solution towards the translational execution of ALF through mechanisms able to transform ALF action code first into intermediate concepts and thereby to the target language.

Also in this task we had to define some constraints in order to achieve de-

terminism in our code generation process. In fact, in its specification, ALF provides three levels of syntactical conformance, namely *minimum*, *full*, and *extended*. In our solution we provide a solution targeting the minimum conformance and we provide translation of most of the entailed concepts (see Appendix C). This choice is motivated by the fact that the set of translated concepts, although limited if considering the expressiveness provided by ALF, reflects the ones which have a correspondence in the target language (and target domain).

Research challenge 3: ***"Define and implement an automated process to enable the back-propagation of monitoring results to design models".***
The main goal of the round-trip approach is to aid the developer in assessing, at modelling level, the preservation of those properties defined in the design model and monitored at runtime. Therefore, while the generation process implements the forward translation from design model to executable code, a backward transformation process is needed to propagate results coming from code execution monitoring back to the design model. More specifically, appropriate means had to be defined and implemented for (i) generating explicit trace links between design model elements and code, (ii) monitoring the generated code, and (iii) back-propagating the gathered values to the design model.

The solution we propose for this challenge encompasses the definition and implementation of intermediate modelling structures for breaking down the back-propagation into smaller and reusable steps as well as the definition and implementation of the transformations in charge of performing the injection of values all the way up from the monitoring artefacts to the design model. Moreover, in order to enable back-propagation activities we needed to define and store precise traceability information along with the code generation process; this has been achieved through the definition of a specific enhanced traceability metamodel (i.e., back-propagation metamodel) as well as ad-hoc transformation rules for generating explicit trace links when code is generated from design models.

In the literature we could not find any documented evidence of similar attempts operating vertically from models to code and back for preservation of system properties. Anyhow, approaches that operate horizontally (model to model or code to code) in terms of abstraction levels can be found even though full automation is usually not targeted. Since we focus on embedded systems, it was crucial for our approach to provide vertical propagation of analysis results gathered at system implementation level back to design models for better understanding of those EFPs that cannot be accurately predicted at higher

levels of abstraction. Moreover, we supply an automated mechanism for back-propagation features in order for the developers to skip overcomplicated and error-prone manual inspections of the code.

In any case, the developer has control over design models and generated code and therefore, by modifying any of them, she may cause inconsistencies during the back-propagation phase and hence jeopardise the reliability of the back-propagated values. That is the reason for which, in order for the proposed approach to guarantee that gathered information is correctly and consistently back-propagated to the design models, generated code is not meant to be manually edited, and once the model is edited, the previously generated code is considered as obsolete and therefore re-generated in compliance to the current model version.

Research challenge 4: ***"Demonstrate how the round-trip approach can be employed to guide engineering decisions based on back-propagated EFP values".***
In our solution to code generation we provide two options: (1) highly resource efficient (in terms of inter-system communications) single process applications and (2) multi process configurations to run in parallel on multicores. Thanks to this variability at deployment level, we were able to demonstrate how the round-trip approach can be employed to guide engineering decisions at modelling level based on measurements gathered at system implementation level; focus was on deployment assessment.

While there is evidence of approaches that target the identification of performance problems and their fix at implementation level, our approach introduces the possibility to assess deployment goodness at a more abstract modelling level still employing values computed at lower abstraction levels (runtime). Once again, this characteristic aims at relieving the developer from complex code inspections and error-prone manual code editing.

11.2 General Issues

Interplay of MDE and CBSE. In this research work we exploited the increasingly popular synergy between MDE and CBSE for modelling, generating and analysing applications in the embedded domain. While some advantages of those approaches as well as their interplay have been described in Chapter 1, in this section we highlight some of the issues coming from their interwoven employment that are often omitted by MDE supporters. On the one hand, one

of the goals of MDE is to generate the final application from the design models, and, as in our case, sometimes the target is the production of the full-fledged application that can be executed more or less directly after its generation. On the other hand, CBSE lays its foundations on the notion of component as a replaceable entity and that software is more and more built by *reusing* pre-existing units (e.g., commercial off-the-shelf (COTS)) [2].

The word "reuse" here may create a conflict between the full generation of MDE and the reuse of components, and thereby legacy code, of CBSE. So what do we do in such a situation, regenerate everything or reuse existing code? The answer probably resides in the middle. In fact, for analysability reasons as well as consistency between artefacts, guarantee and correctness-by-construction of the generated code, the favourable choice would be to regenerate the code according to the actual development process and constraints. On the opposite, due to safety, security and certification reasons, components might have to be kept as they are including related code. Combining these reasons, a possible solution could be to model the components to be reused as "blackboxes" and treat them as stubbed and therefore implemented by means of external binaries or libraries to be included at code level. In order to achieve this, proper interfaces between these blackboxes and the rest of the system shall be defined and implemented according to specific constraints, such as *contracts* [70], for defining what the component maintains, what it expects from the surroundings and what it guarantees both functionally and extra-functionally.

In our research work we did not exploit CBSE in its broader sense, but rather employed its design pattern (as defined in the UML superstructure) for encapsulation and concerns-separation reasons. That is to say that a component might have been modelled as a class or another modelling concept without affecting the applicability of the solution. Thus, in our case code is generated for the whole model hence leaving open the possibility to interact with existing binaries by treating them as libraries at model (especially action code) level. Anyhow, an interesting future direction could be to approach CBSE in its foundational sense through the introduction of the notion of reusable components as well as the definition of the means to describe interactions between legacy components and the surrounding system.

Model Versioning. One of the main motivations behind the kind of research work presented in this thesis is to strengthen the willingness from industry to favour MDE to the detriment of error-prone and costly code-centric approaches. In order to do that, automation and support to the developer shall be provided at least at the same level available for code-centric approaches. In this

thesis we described in detail our solution for model-driven development of embedded systems with focus on properties preservation through full-fledged code generation, execution monitoring and back-propagation from code to model.

Nevertheless, many related issues remain open. Let us consider, for instance, versioning of development artefacts. Version Control Systems (VCSs) have been proven successful in code versioning, but they are only partially suitable for handling versioning in the modelling domain. In fact, differences and conflicts between versions of a same artefact are usually detected at file-level through line-oriented text comparison. However, even if taking into account model XMI serialisations (standardised for UML) as [71], the abstraction mismatch between text and models may lead to erroneous detection of differences and hence conflicts [72, 73]. Therefore, a number of research works have been and still are devoted to versioning models [74] and metamodels [75] at the appropriate level of abstraction, that is to say at modelling level rather than at underlying textual level, advancing the state of the art in (meta)model differencing, versioning, and related co-evolution problems.

While in this research work we did not focus on versioning problems, future extensions could entail versioning features in order to enable distributed development while keeping intact the integrity of the guarantee-oriented nature of our approach. In order to do this, our contributions related to model versioning [76, 77, 78], which are not part of this thesis, would be exploited.

UML Profiles or Domain-Specific Modelling Languages. The approach presented in this thesis aims at generally showing that MDE should contemplate the possibility that, in many development processes the task of code generation may be a transitional step rather than a final non-coming back one. This applies in particular when the focus is on preservation of system properties and when some EFPs cannot be predicted with accuracy at early stages. Studying the literature and from the experiences matured during this research work, in several applicative domains we can identify cases in which such a need arises. This makes us believe that the idea of enforcing full automation in generating code as well as the ability to employ data gathered at system implementation level and back-propagated for optimising models (and thereby code) rather than employing them for optimising code with manual activities will (and at some extend already does) draw significant interest in the community.

The basic technologies on which we built our approach upon were modelling languages and model transformations. More specifically regarding the languages to design the system under development as well as to host back-propagated information, we employed the CHESS-ML which leverages on a

subset of UML, ALF and MARTE. The idea behind CHESS-ML is to provide a language which is not as general-purpose as UML nor tailored to one single domain. Rather it should be able to provide means for diverse domains with common needs (i.e., automotive, aerospace, telecommunication, railways) in terms of EFPs and constraints to exploit it for the development of embedded (real-time) systems.

In the MDE community, the usage of UML and its profiling mechanisms for the definition of domain-specific languages fires up endless debates due to several reasons [79]. Being one of the main goals to provide a "unified", therefore as generic as possible, language, many concepts as well as different nuances of them have been introduced as part of language throughout the years.

On the one side this can be positively seen as a way to represent a wide range of concepts with no major limitation, but on the other side one could argue that practically only very few concepts at a time are employed in a development process, making the language way too expressive and heavy in both its abstract and concrete syntax than actually needed in many cases. At this point a possible solution could be the employment of profiling mechanisms to tailor the UML to a specific domain and somehow circumscribe its expressiveness and focus on selected aspects. Also in this case a debate usually arises; does a UML profile really incarnate the notion of domain-specific modelling language (DSML)? The answer depends on the perspective.

According to UML backers, exploiting profiling for tailoring UML to a specific domain should bring along several advantages, first of all the possibility to systematically introduce new language elements without having to re-create the whole modelling ecosystem (including model transformations and model editors), but also the prevention of metamodel pollution as well as the possibility to reuse existing tools and analysis mechanisms defined for UML and its profiles. On the other side, supporters of DSMLs as brand new languages defined for and dedicated to a specific domain struggle to agree with the vision of a UML profile as such. In fact, extending a large, general-purpose language (as UML), although tempting given the possibility to exploit existing language's syntax and semantics, does not diminish the fact that such a language is usually too generic and broad for any specific domain [80]. The work of adding concepts and semantics to a large existing language seems often to be a harder work than starting from scratch [79]. At the same time, when defining a DSML from scratch one has to pay attention in the balance between generality and specificity, since too much emphasis on domain-specific aspects may make further changes or extensions to the DSML overcomplicated or even infeasible.

In our experience we could profit from some of the features provided by the UML family, such as ALF for a model-aware definition of complex behaviours, and MARTE for modelling EFPs and deployment concepts as well as UML tools (i.e., Papyrus). At the same time we experienced drawbacks, like the fact that the modelling ecosystem, especially model transformations and editors, actually needs adaptation, sometimes quite tedious, when UML existing concepts are stereotyped or when the profile itself undergoes modifications. Our feeling is that the choice of defining a DSML or exploit UML profiling is not an easy task and should be taken without preconceptions considering a wide set of variables such as size of the domain, end user's preferences and willingness in eventually changing current toolset (in case of industrial processes), expected lifetime of the language to be defined, just to mention a few.

Full Code Generation: Is It Really Worth the Effort? Usually in order to achieve full code generation more or less extensive amount of information needs to be modelled. Usually modelling activities lead to the so called *accidental complexity* [81], meant as an additional effort introduced in the development process by activities which are not directly essential to the solution of the problem. This would mean that, in some cases, the effort required by modelling activities to reach the level of details necessary for full generation of code is equal or even higher than the effort that would be needed to implement the application by hand.

In our approach we target a full code generation since the overall goal was to provide support for preservation of system properties. Allowing the possibility to edit the system be means of manual fixes at code level would in fact break the consistency between models and generated code, thus the results of any analysis performed at modelling level could become invalid at code level and the other way around. But what about accidental complexity in our case? The fact that the employed modelling language (i.e., CHESS-ML) restricts to a minimum the set of UML-MARTE concepts available to the user through profiling and that the CHESS methodology itself provides means to drive[1] the user in the modelling activities helped us in containing accidental complexity. Anyhow, in comparison to traditional model-driven processes where only skeletons are meant to be generated, the additional effort required by our approach is represented by the definition of behaviours through action code. In any case, the

[1]The CHESS methodology drives the user in modelling activities mainly in two ways: (1) separated design views defined at metamodel level and enforced by constraints at tool level, and (2) constraints, on-the-fly checks and validators in editors and palettes at tool level.

employment of a fully model-aware formalism (i.e., ALF) provides a number of benefits, described in Chapter 7.

Finally, we believe that code generators are very useful tools as long as accidental complexity is somehow limited. This is usually easier to achieve when entailing a domain-specific language that naturally incarnates the characteristics of a specific domain, while more intricate when employing general-purpose languages (for the reasons mentioned in the previous paragraph). From our experience, a solution is feasible even in the latter case as long as we delimit context, and usually domain, at modelling level (through constraints at both language and tool level) in which the code generator is meant to be used.

Chapter 12

Conclusions and Future Work

Growth of power and miniaturization of modern systems, especially in the embedded domain, are inevitably correlated to an increment of their complexity. In turn, such an increment may run against manufacturers' common goals of reducing costs and time-to-market. In this scenario, code-centric approaches are slowly but continuously loosing their predominant position in development processes due to several reasons, ranging from error-proneness to the difficulty of testing the system under development at very early stages and thereby the costly and time consuming task of testing and adjusting the system at product level. It has been repeatedly shown over the last 15 years that abstraction, separation of concerns, smart reuse, early analysis and automation are among the keys to simplify the development and therefore tackle modern systems' ever-increasing complexity. Among them, model-driven engineering and component-based software engineering have grown consideration for their ability to provide these key features and in this research work we exploit a combination of model-driven development mechanisms together with component-based design pattern.

One of the core goals of a model-driven approach is the provision of automated code generation from design models; however, this goal is too often seen as the very final step of the development process. This means that while exploiting new powerful development mechanisms, we are unconsciously going back to a waterfall-like process, were the generated implementation ratifies the end of the development. But, what happens for those extra-functional prop-

erties that could not be analysed at earlier stages? The answer seems to be simple: generated code is executed, properties are measured and, if not satisfactory, code is edited accordingly. In other words, we would void the benefits we achieved through abstraction, separation of concerns, early analysis and automation by, once again, costly and time consuming manual activities with possible introduction of errors and broken consistency among development artefacts (i.e., models and generated code).

As possible answer to this issue, the outcome of this research work is a model-driven technique that aids the preservation of system properties from models to generated code by introducing the novel notion of *back-propagation* across different abstraction levels (i.e., from runtime to model). The proposed solution is represented by a round-trip approach consisting of four steps: (1) system modelling, (2) automatic full-fledged code generation, (3) monitoring of the execution of the generated implementation and computing extra-functional properties of interest, and finally (4) back-propagation of monitoring results to the design model.

In this thesis we exposed the details concerning both problems, defined as a set of research challenges, and solutions, defined as a set of thesis contributions. Moreover we provided information concerning the validation of the approach both in-house and in industrial settings. A discussion of the contributions in relation to the research challenges they attempt to solve is provided together with a discussion on diverse general aspects related to the techniques and technologies that have been exploited to achieve the solutions.

Enhancement of Back-propagation. Future research directions could encompass the extension of the proposed approach by taking into account management and evaluation of properties like safety and security, which usually differ from properties measurable by means of computed values. Moreover, new generation EFPs, such as energy-related properties, could be taken into consideration for future evaluations and possible extensions of the provided solution.

Dealing with multicore platforms, where EFPs may vary depending on the execution instance we refer to, back-propagation capabilities could be enhanced to entail incremental decoration of the design model with multiple values, for the same EFP, gathered from the monitoring of different execution instances. Extra-functional values in relation to a specific execution instance (and platform configuration) can be helpful for the developer to, e.g., analyse whether and how different configurations or even simply different execution instances of a same configuration affect EFPs. In order to enable this kind of

feature appropriate concepts at modelling level for hosting the related information in a structured manner should be identified, if they exist, and refined, if they do not fully provide support for our needs.

Additionally, given the ability of the approach of supporting different levels of granularity both for traceability and back-propagation, a possible enhancement could also be the possibility for the developer to select the wished level (or levels) of granularity at modelling level at the beginning of a round-trip iteration.

Model Execution and Simulation. In the development of complex systems, the ability to simulate prototypes already at very early design phases has gained increasing attention [82]. Diverse simulation environments based on mathematical foundations have been exploited to deal with the interaction between system under development and surrounding environment [82, 83], while others transform the system specification into formal representations to emulate the system's behaviour [84].

With the introduction of model-driven engineering and its abstractive capabilities, a new opportunity of employing design models for simulation has bloomed. In the case of UML, the formalisation of a precise execution semantics for a subset of the language in terms of the fUML, together with the introduction of an action language (i.e., ALF) compliant to it for the specification of complex behaviours, gives the possibility to build simulation environments for UML models.

One interesting and challenging future work could be the exploitation of these formalisms, together with the specification of a more detailed platform model, to define a model simulation and execution environment able to show control and data flows by means of animation on models based on a model interpreter. This kind of feature could allow testing and validation of relevant system properties before generating the implementation by simulating it, in a way similar to monitoring of code execution but at an earlier stage. Such an environment may help in decreasing the number of iterations of the round-trip approach needed by the developer to reach the wished level of system quality.

Code Generation and Multi-platform Runtime Environment. Thanks to satisfactory results achieved in running the industrial case-study, our approach has been recognised as promising and possibly useful as compliment to currently used tools. Particularly relevant would be the possibility to extend the current approach to provide a standalone runtime execution component to be

used to deploy models at runtime (motivations for this have already been depicted in the previous paragraph). The overall idea would be to enable the deployment of design models onto platforms like Linux and OSE. Focus would be on providing a runtime component which produces code optimised for multicore.

In order to achieve this, the current code generator shall be enhanced in different directions. One of them is to take into account hierarchical nested states as well as orthogonal regions. For hierarchical nested states (or hierarchical state-machines) we mean a decomposition which can be seen as an exclusive-OR operation applied to states meaning that, if a system is in a superstate (OR-state), then it is in one of the substates. For orthogonal regions we mean the possibility of having AND-decomposition. This means that a composite state may contain several orthogonal regions and that being in this state entails being in all its orthogonal regions at the same time. The ability of the code generator to account these concepts brings along several challenges especially regarding the definition of execution patterns for orthogonal behaviours in relation to the deployment platform.

Moreover, in this work we provide a preliminary solution for multicore-aware generation of code, though leaving most of the multicore-related decisions to the operating system (i.e., OSE). In this sense we just opened Pandora's box since many details regarding memory, tasks division and allocation, should be both modelled as well as properly taken into account by the transformation process to enable multicore *optimised* generation of code; some work in this direction has already begun.

Heterogeneous Applications. In several domains (e.g., medical, automotive, aerospace) embedded systems are expected to process massive amounts of data, even in real-time. Aiding in fulfilling these expectations, the development of hardware technologies towards heterogeneous configurations makes embedded systems able to handle, e.g., very high input data rates. A typical scenario would be represented by input data coming into a multicore socket, which in turn may exploit one or more GPUs as coprocessors for parallel processing of large blocks of data [85]. Such a technology shift from homogeneous to heterogeneous hardware configurations raises a number of new research issues on both the modelling and coding of embedded systems.

On the one hand, the introduction of heterogeneity gives us the possibility to enable faster computation and generally increase the performances of the generated implementation. On the other hand, it adds an additional level of complexity in the hardware and deployment configuration and therefore com-

plicates the code generation process. Due to the fact that CPUs and GPUs employ different formalisms and mechanisms for code execution, as well as different programming languages, the transformation process has to be enhanced in order to be able to map model entities to code artefacts written in different target languages (e.g., `C++` to be run on CPUs and `OpenCL` or `CUDA` to be run on GPUs) [85]. Moreover, the transformation will have to generate the communication code needed for the interaction between CPUs and GPUs.

Towards this goal we already achieved some preliminary results in generating heterogeneous applications [85]. Since thorough validation of the code generator as well as monitoring and back-propagation features for these scenarios were not in place yet, the results have been left out from this thesis.

Bibliography

[1] J. Bezivin. On the unification power of models. *Software and Systems Modeling*, 4:171–188, 2005.

[2] I. Crnkovic. Component-based software engineering for embedded systems. In *Proceedings of International Conference on Software Engineering (ICSE)*, pages 712–713. ACM, 2005.

[3] R. Land, J. Carlson, S. Larsson, and I. Crnkovic. Project Monitoring and Control in Model-driven and Component-based Development of Embedded Systems – The CARMA Principle and Preliminary Results. In *Proceedings of International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE)*, pages 253–258, 2010.

[4] Object Management Group. Model Driven Architecture. `http://www.omg.org/cgi-bin/doc?omg/00-11-05.pdf`, November 2000.

[5] D. Cancila, R. Passerone, T. Vardanega, and M. Panunzio. Toward Correctness in the Specification and Handling of Non-Functional Attributes of High-Integrity Real-Time Embedded Systems. *IEEE Transactions on Industrial Informatics*, 6(2):181–194, May 2010.

[6] S. E. Chodrow, F. Jahanian, and M. Donner. Monitoring and debugging of distributed real-time systems. In Jeffrey J. P. Tsai and Steve J. H. Yang, editors, *Run-time monitoring of real-time systems*, pages 103–112. IEEE Computer Society Press, Los Alamitos, CA, USA, 1995.

[7] N. Siegmund, M. Rosenmuller, M. Kuhlemann, C. Kastner, and G. Saake. Measuring Non-Functional Properties in Software Product Line for Product Derivation. In *Proceedings of Asia-Pacific Software Engineering Conference (APSEC)*, pages 187 –194, 2008.

[8] S. Kent. Model Driven Engineering. In *Proceedings of International Conference on Integrated Formal Methods (IFM)*, 2002.

[9] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, pages 621–645, 2006.

[10] Object Management Group. UML Superstructure Specification V2.3. `http://www.omg.org/spec/UML/2.3/Superstructure/PDF/`, 2011.

[11] N. Delgado, A.Q. Gates, and S. Roach. A taxonomy and catalog of runtime software-fault monitoring tools. *IEEE Transactions on Software Engineering*, 30(12):859 – 872, 2004.

[12] R. Chapman. Correctness by construction: a manifesto for high integrity software. In *Proceedings of Australian workshop on Safety critical systems and software (SCS)*, pages 43–46, 2005.

[13] CHESS Consortium. CHESS Project Website. `http://www.chess-project.org/`, February 2009.

[14] A. Cicchetti, F. Ciccozzi, S. Mazzini, S. Puri, M. Panunzio, A. Zovi, and T. Vardanega. CHESS: a model-driven engineering tool environment for aiding the development of complex industrial systems. In *Proceedings of International Conference on Automated Software Engineering (ASE)*, pages 362–365. ACM, 2012.

[15] B. Selic. Unified Modeling Language (UML). In *Wiley Encyclopedia of Computer Science and Engineering*. 2008.

[16] Object Management Group. Systems Modeling Language (SysML), Version 1.3. `http://www.omg.org/spec/SysML/1.3/PDF/`, 2012.

[17] Object Management Group. UML Profile For MARTE: Modeling And Analysis Of Real-Time Embedded Systems, v1.1. `http://www.omg.org/spec/MARTE`, June 2011.

[18] S. Gérard, C. Dumoulin, P. Tessier, and B. Selic. Papyrus: A UML2 Tool for Domain-Specific Language Modeling. In *Model-Based Engineering of Embedded Real-Time Systems*, pages 361–368, 2007.

[19] F. Budinsky, D. Steinberg, E. Merks, R. Ellersick, and T.J. Grose. *Eclipse Modeling Framework*. Addison Wesley, 2003.

[20] Object Management Group. Action Language For FoundationalUML - ALF. `http://www.omg.org/spec/ALF/`, Oct 2010.

[21] J.M. Nahman. *Dependability of Engineering Systems: Modeling and Evaluation*. Springer. 2002.

[22] B. Gallina and S. Punnekkat. Fi4fa: A formalism for incompletion, inconsistency, interference and impermanence failures analysis. In *Proceedings of International workshop on Distributed Architecture modeling for Novel Component based Embedded systems (DANCE)*, pages 493–500, 2011.

[23] M. Bordin, M. Panunzio, and T. Vardanega. Fitting Schedulability Analysis Theory into Model-Driven Engineering. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 135–144, 2008.

[24] Enea. The Architectural Advantages of Enea OSE in Telecom Applications. `http://www.enea.com/software/solutions/rtos/`.

[25] M. Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Languange*. Addison-Wesley Professional, 2004.

[26] M. Alras, P. Caspi, A. Girault, and P. Raymond. Model-Based Design of Embedded Control Systems by Means of a Synchronous Intermediate Model. In *Proceedings of International Conference on Embedded Software and Systems (ICESS)*, pages 3–10, may 2009.

[27] M. Usman, A. Nadeem, and Tai hoon Kim. UJECTOR: A Tool for Executable Code Generation from UML Models. In *Proceedings of International Conference on Advanced Software Engineering and Its Applications (ASEA)*, pages 165–170, 2008.

[28] T.G. Moreira, M.A. Wehrmeister, C.E. Pereira, J.-F. Petin, and E. Levrat. Automatic code generation for embedded systems: From UML specifications to VHDL code. In *Proceedings of International Conference on Industrial Informatics (INDIN)*, pages 1085–1090, 2010.

[29] F. Ciccozzi, A. Cicchetti, and M. Sjödin. Round-trip Support for Extra-Functional Property Management in Model-Driven Engineering of Embedded Systems. *Information and Software Technology*, 55:1085–1100, 2013.

[30] N. Aizenbud-Reshef, B. T. Nolan, J. Rubin, and Y. Shaham-Gafni. Model traceability. *IBM Systems Journal*, 45:515–526, 2006.

[31] G. K. Olsen and J. Oldevik. Scenarios of traceability in model to text transformations. In *Proceedings of European Conference on Modelling Foundations and Applications (ECMDA-FA)*, pages 144–156. Springer-Verlag, 2007.

[32] R. Händel, M.N. Huber, and S. Schröder. *ATM Networks Concepts, Protocols, Applications*. Addison-Wesley, 1994.

[33] S. Boyko, R. Dvorak, and A. Igdalov. The Art of Model Transformation with Operational QVT. `http://www.eclipse.org/m2m/qvto/doc/EclipseCon_2009.ppt`, March 2009.

[34] C. Bock. UML 2 Composition Model. *Journal of Object Technology*, 3(10):47–74, 2004.

[35] F. Chauvel and J.-M. Jezequel. Code Generation from UML Models with Semantic Variation Points. In *Proceedings of International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, volume 3713 of *Lecture Notes in Computer Science*, pages 54–68. Springer Berlin Heidelberg, 2005.

[36] F. Fleurey, Z. Drey, D. Vojtisek, C. Faucher, and V. Mahé. Kermeta Language, Reference Manual. `http://www.kermeta.org/docs/KerMeta-Manual.pdf`, 2006.

[37] E. Börger, A. Cavarra, and E. Riccobene. A precise semantics of UML state machines: making semantic variation points and ambiguities explicit. In *Proceedings of Workshop on Semantic Foundations of Engineering Design Languages (SFEDL)*, pages 1–20, 2002.

[38] I. Oliver and V. Luukala. On UML's Composite Structure Diagram. In *Proceedings of Workshop on System Analysis and Modelling (SAM)*, 2006.

[39] A. Cuccuru, S. Gérard, and A. Radermacher. Meaningful Composite Structures. In *Proceedings of International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, volume 5301 of *LNCS*, pages 828–842. Springer, 2008.

[40] I. Ober and I. Dragomir. Unambiguous UML Composite Structures: The OMEGA2 Experience. In *Theory and Practice of Computer Science (SOFSEM)*, volume 6543 of *LNCS*, pages 418–430. Springer, 2011.

[41] A. Radermacher, A. Cuccuru, S. Gerard, and F. Terrier. Generating execution infrastructures for component-oriented specifications with a model driven toolchain: a case study for MARTE's GCM and real-time annotations. In *Proceedings of International Conference on Generative Programming and Component Engineering (GPCE)*, volume 45, pages 127–136. ACM, 2009.

[42] M. Panunzio and T. Vardanega. On Component-Based Development and High-Integrity Real-Time Systems. In *Proceedings of International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 79–84, 2009.

[43] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[44] M. Brun, J. Delatour, and Y. Trinquet. Code Generation from AADL to a Real-Time Operating System: An Experimentation Feedback on the Use of Model Transformation. In *Proceedings of International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 257–262, april 2008.

[45] M. Fredj, A. Radermacher, S. Gerard, and F. Terrier. eC3M: Optimized model-based code generation for embedded distributed software systems. In *Proceedings of International Conference on New Technologies of Distributed Systems (NOTERE)*, pages 279–284, june 2010.

[46] W. Haberl, M. Tautschnig, and U. Baumgarten. *Generating Distributed Code From COLA Models*, volume 33 of *Lecture Notes in Electrical Engineering*, chapter 20. Springer, March 2009.

[47] J. Vidal, F. de Lamotte, G. Gogniat, P. Soulard, and J.-P. Diguet. A co-design approach for embedded system modeling and code generation with UML and MARTE. In *Proceedings of Design, Automation and Test in Europe (DATE)*, pages 226–231, april 2009.

[48] C. Xi, L. JianHua, Z. ZuCheng, and S. YaoHui. Modeling SystemC design in UML and automatic code generation. In *Proceedings of Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 932–935. ACM, 2005.

[49] Q. Long, Z. Liu, X. Li, and H. Jifeng. Consistent code generation from UML models. In *Proceedings of Australian Software Engineering Conference (ASWEC)*, pages 23–30, 2005.

[50] Object Management Group. Foundational Subset For Executable UML Models (FUML). `http://www.omg.org/spec/FUML/1.1/`, Last Accessed: July 2013.

[51] H. A. Müller, J. H. Jahnke, D. B. Smith, M.-A. Storey, S. R. Tilley, and K. Wong. Reverse engineering: a roadmap. In *Proceeding of Conference on The Future of Software Engineering (ICSE)*, pages 47–60. ACM, 2000.

[52] Linux Die. Linux 2.6.9 Manual. `http://linux.die.net/man/2/getrusage`, 2006.

[53] I. Wenzel, R. Kirner, B. Rieder, and P. Puschner. Measurement-based worst-case execution time analysis. In *Proceedings of International Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS)*, pages 7–10. IEEE, 2005.

[54] F. Ciccozzi, M. Saadatmand, A. Cicchetti, and M. Sjödin. An Automated Round-trip Support Towards Deployment Assessment in Component-based Embedded Systems. In *Proceedings of International Symposium on Component-Based Software Engineering (CBSE)*. ACM, 2013.

[55] Z. Navabi, S. Day, and M. Massoumi. Investigating Back Annotation of Timing Information into Dataflow descriptions. In *Proceedings of VHDL International User Forum*, pages 185–195, 1992.

[56] G. Mahadevan and J. R. Armstrong. Investigating Back Annotation of Timing Information into Dataflow descriptions. In *Proceedings of VHDL Interational User Forum (VIUF)*, 1995.

[57] Á. Hegedüs, G. Bergmann, I. Ráth, and D. Varró. Back-annotation of Simulation Traces with Change-Driven Model Transformations. In *Proceedings of International Conference on Software Engineering and Formal Methods (SEFM)*, pages 145–155, 2010.

[58] E. Guerra, D. Sanz, P. Díaz, and I. Aedo. A transformation-driven approach to the verification of security policies in web designs. In *Proceedigs of International Conference on Web Engineering (ICWE)*, pages 269–284, Berlin, Heidelberg, 2007. Springer-Verlag.

[59] A. Wall, J. Kraft, J. Neander, C. Norström, and M. Lembke. Introducing Temporal Analyzability Late in the Lifecycle of Complex Real-Time Systems. In *Proceedings of International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. Springer Berlin Heidelberg.

[60] M. Saadatmand, A. Cicchetti, and M. Sjödin. Design of adaptive security mechanisms for real-time embedded systems. In *Proceedings of International Symposium on Engineering Secure Software and Systems (ESSoS)*, pages 121–134. Springer-Verlag, 2012.

[61] M. Saadatmand, M. Sjödin, and N. U. Mustafa. Monitoring Capabilities of Schedulers in Model-Driven Development of Real-Time Systems. In *Proceedings of International Conference on Emerging Technologies & Factory Automation (ETFA)*, 2012.

[62] J. Huselius and J. Andersson. Model Synthesis for Real-Time Systems. In *Proceedings of European Conference on Software Maintenance and Reengineering (CSMR)*, pages 52–60. IEEE Computer Society.

[63] H. Koziolek. Performance evaluation of component-based software systems: A survey. *Journal of Performance Evaluation*, 67(8):634–658, 2010.

[64] S. Yacoub. Performance Analysis of Component-Based Applications. In *Software Product Lines*, LNCS, pages 299–315. Springer Berlin Heidelberg, 2002.

[65] A. Mos and J. Murphy. A framework for performance monitoring, modelling and prediction of component oriented distributed systems. In *Proceedings of International Conference on Performance Engineering (WOSP)*, pages 235–236. ACM, 2002.

[66] A. Diaconescu and J. Murphy. Automating the performance management of component-based enterprise systems through the use of redundancy. In *Proceedings of International Conference on Automated Software Engineering (ASE)*, pages 44–53. ACM.

[67] T. Parsons and J. Murphy. Detecting Performance Antipatterns in Component Based Enterprise Systems. *Journal of Object Technology*, pages 55–91, 2008.

[68] TESTEJB - A Measurement Framework for EJBs. In *Proceedings of International Conference on Component-Based Software Engineering (CBSE)*, LNCS, pages 294–301. Springer Berlin Heidelberg, 2004.

[69] N. Katanic and M. Perse. Application of CHESS Methodology: A Telecom Use Case Study. In *Proceedings of International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*, 2012.

[70] Z. Liu, H. Jifeng, and X. Li. Contract oriented development of component software. In *Exploring New Frontiers of Theoretical Informatics*, pages 349–366. Springer, 2004.

[71] H. Oliveira, L. Murta, and C. Werner. Odyssey-VCS: a flexible version control system for UML model elements. In *Proceedings of International Workshop on Software Configuration Management (SCM)*, pages 1–16. ACM, 2005.

[72] M. Alanen and I. Porres. Difference and Union of Models. In *UML 2003 - The Unified Modeling Language*, volume 2863, pages 2–17, 2003.

[73] D.S. Kolovos, R.F. Paige, and F.A.C. Polack. Model comparison: a foundation for model composition and model transformation testing. In *Proceedings of International Workshop on Global Integrated Model Management (GaMMa)*, pages 13–20. ACM Press, 2006.

[74] K. Altmanninger, M. Seidl, and M. Wimmer. A survey on model versioning approaches. *International Journal of Web Information Systems*, 5(3):271–304, 2009.

[75] B. Meyers and H. Vangheluwe. A framework for evolution of modelling languages. *Science of Computer Programming*, 76(12):1223–1246, December 2011.

[76] A. Cicchetti, F. Ciccozzi, and T. Leveque. Supporting Incremental Synchronization in Hybrid Multi-View Modelling. In *Proceedings of International Workshop on Multi-Paradigm Modeling (MPM)*. Springer, December 2011.

[77] A. Cicchetti, F. Ciccozzi, and T. Leveque. A Solution for Concurrent Versioning of Metamodels and Models. *Journal of Object Technology*, August 2012.

[78] F. Ciccozzi and A. Cicchetti. Towards Migration-Aware Filtering in Model Differences Application. In *Proceedings of International Workshop on Models and Evolution (ME)*, October 2012.

[79] S. Kelly and R. Pohjonen. Worst Practices for Domain-Specific Modeling. *IEEE Software*, 26(4):22–29, 2009.

[80] S. Kelly and J.-P. Tolvanen. *Domain-Specific Modeling - Enabling Full Code Generation*. Wiley, 2008.

[81] R. France and B. Rumpe. Model-driven Development of Complex Software: A Research Roadmap. pages 37–54, 2007.

[82] F. Boulanger and C. Hardebolle. Simulation of multi-formalism models with modhel'x. In *Proceedings of International Conference on Software Testing, Verification, and Validation (ICST)*, pages 318–327, 2008.

[83] MathWorks. Simulink Simulation and Model Based Design. `http://www.tufts.edu/~rwhite07/PRESENTATIONS_REPORTS/simulink.pdf`, 2013.

[84] G. Karsai, A. Agrawal, F. Shi, and J. Sprinkle. On the use of graph transformations in the formal specification of model interpreters. *Journal of Universal Computer Science*, 9:1296–1321, 2003.

[85] M. Arora, S. Nath, S. Mazumdar, S. Baden, and D. Tullsen. Redefining the Role of the CPU in the Era of CPU-GPU Integration. *Micro, IEEE*, (99), 2012.

Appendix A

Intermediate Metamodel in Ecore

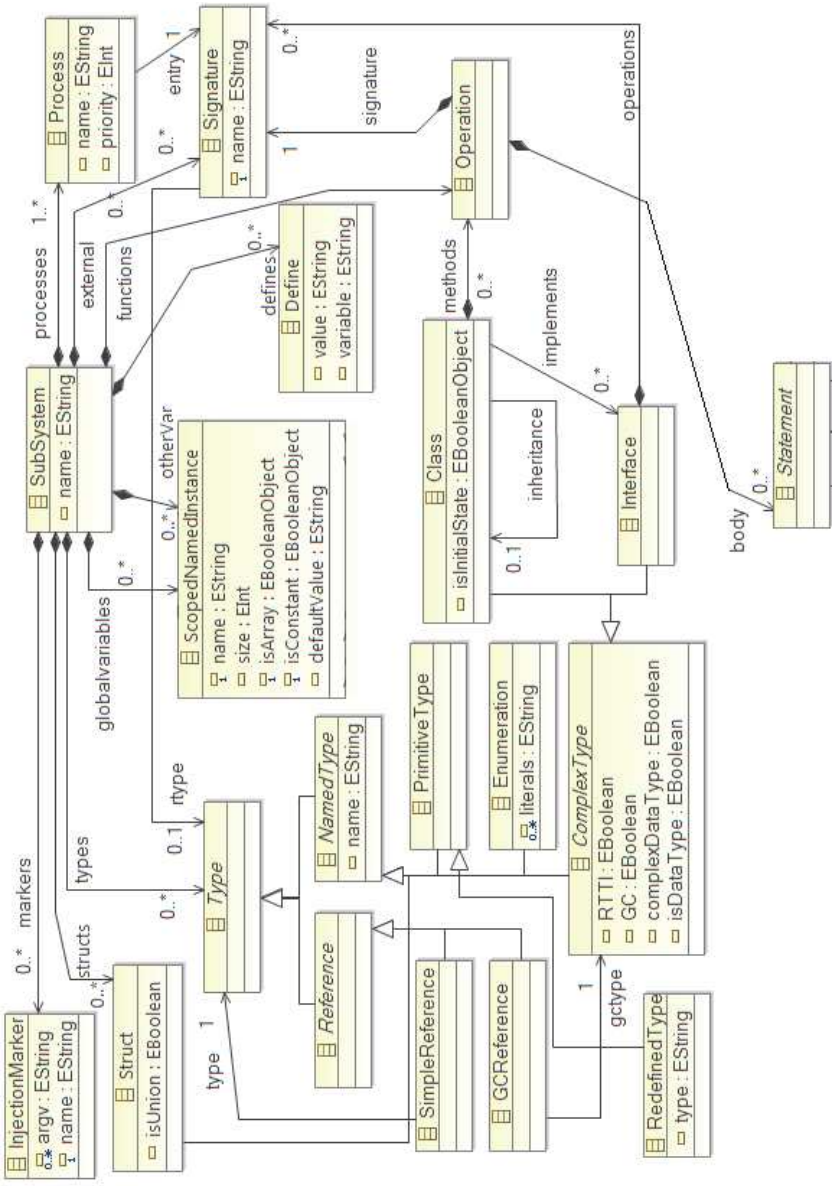


Figure A.1: Subsystem portion of InterMM

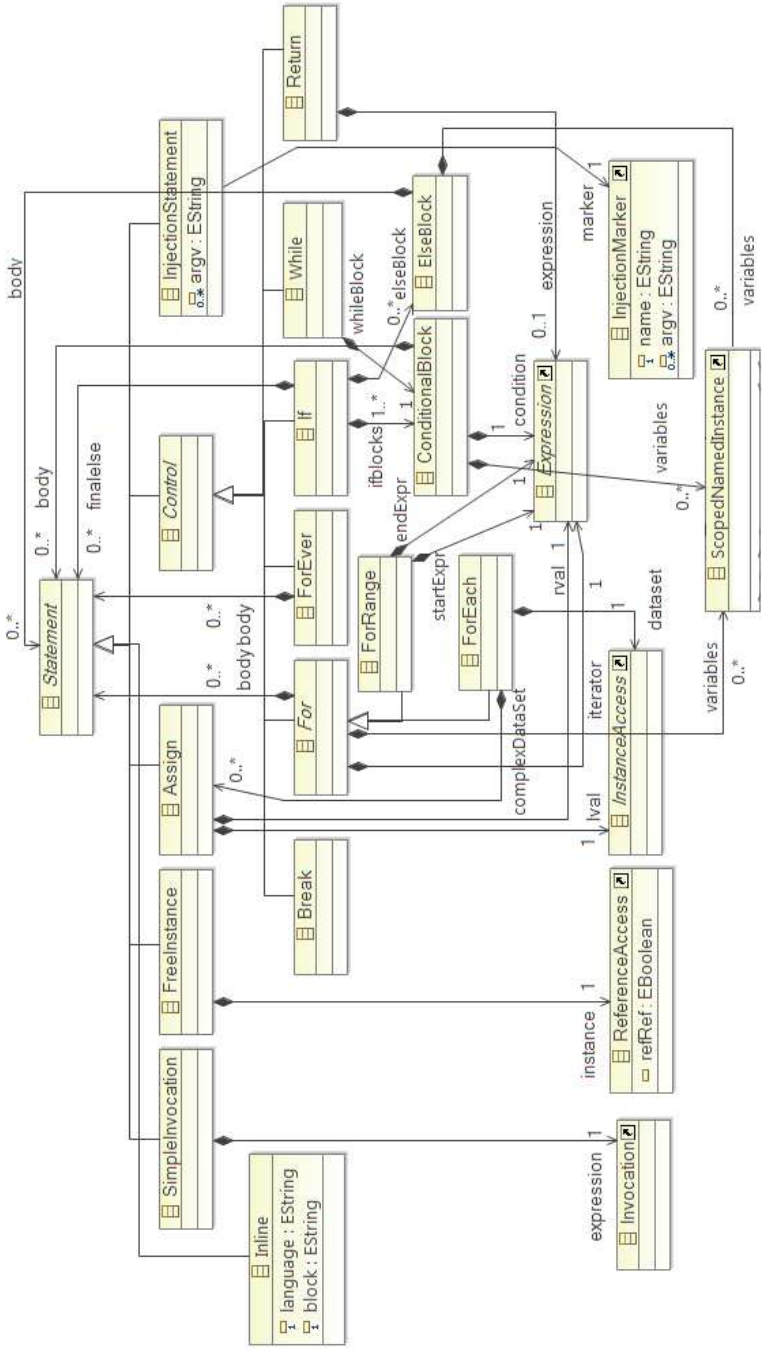


Figure A.2: Statement portion of InterMM

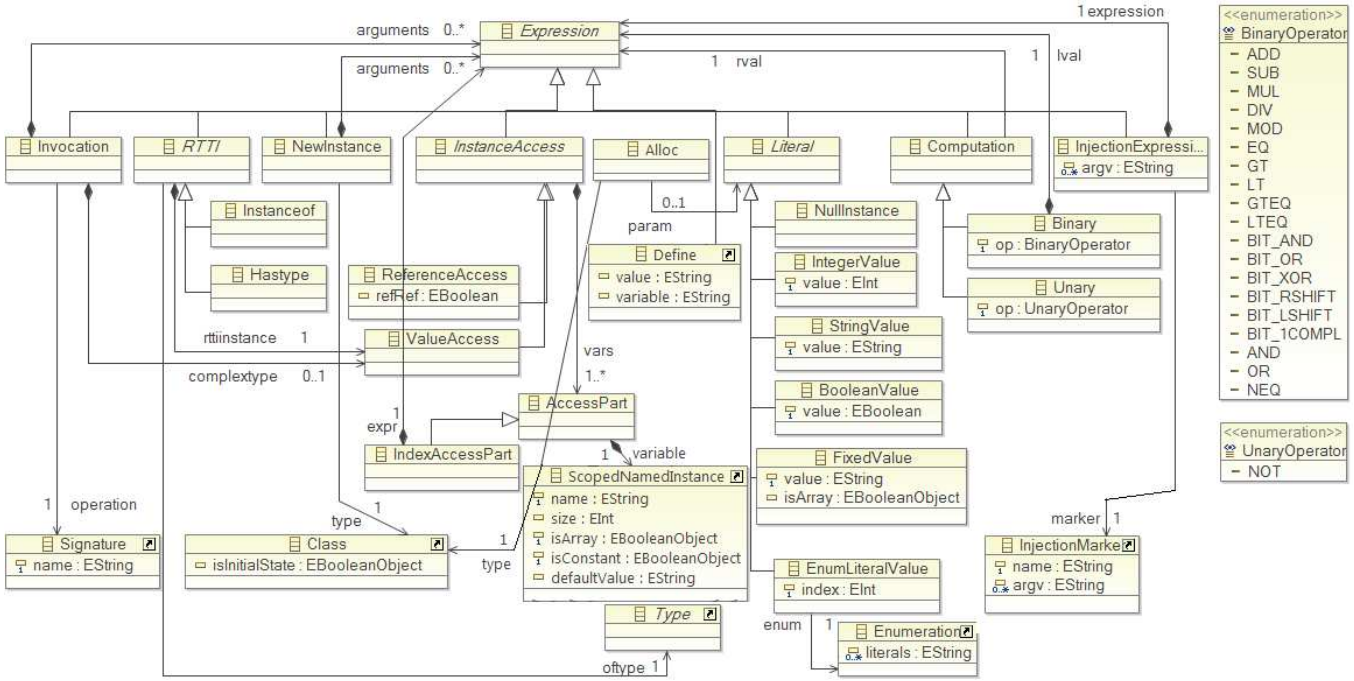


Figure A.3: Expression portion of InterMM

Appendix B

QVTo Transformation for If Statement

```
/* Create the MMOOP::If statement from the ALF model */
helper handleIf(inout smt : ALF::IfStatement){
  var ris = smt.mapIf();
  CURRENT_OPERATION.body += getIf(ris.expr, ris.blocks, ris.blockELSE);
  return null;
}

/**    Return all if conditional expressions, blocks and else block
 *     Else-If conditional expressions and blocks are also included in the
       returned sets */
helper ALF::IfStatement::mapIf() : expr:OrderedSet(MMOOP::Expression),
       blocks:OrderedSet(ALF::Block), blockELSE:ALF::Block {
  var allif = self.sequentialClausses->asOrderedSet().getAllIfStatements();
  var e : OrderedSet(MMOOP::Expression);
  allif.cond->forEach(c){
    var condExp := c.getExpression(false);
    if(not condExp.oclIsUndefined())then{
      e += condExp;
    }endif;
  };
  expr += e;
  blocks := allif.block->asOrderedSet();
  blockELSE := self.finalClause.block;
}

/* Create the MMOOP::If statement */
query getIf(in conds : OrderedSet(MMOOP::Expression), in bodiesEIF :
        OrderedSet(ALF::Block), in bodyElse : ALF::Block) : MMOOP::If{
  var IF := object MMOOP::If{};
  if(conds->size() = bodiesEIF->size())then{
    var index := 0;
    conds->forEach(con){
      index := index + 1;
      var cb := object MMOOP::ConditionalBlock{};
      cb.condition := con;
      handleAlfBlockInIF(bodiesEIF->at(index).oclAsType(ALF::Block),cb);
      IF.ifblocks += cb;
    };
    if(not bodyElse.oclIsInvalid())then{
      var elseBl := object MMOOP::ElseBlock{};
      handleAlfBlockInELSE(bodyElse, elseBl);
      IF.elseBlock := elseBl;
    }endif;
  }endif;
  return IF;
}
```

Figure B.1: Transformation from ALF's *If* statement to InterMM's *If* statement

Appendix C

Coverage of ALF Expressions and Statements

ALF Expressions	Status
Literal Expressions	supported
Name Expressions	supported
this Expressions	supported
Parenthesized Expressions	supported
Property Access Expressions	supported
Invocation Expressions	supported
Tuples	supported
Behavior Invocation Expressions	not supported
Feature Invocation Expressions	not supported
Super Invocation Expressions	not supported
Instance Creation Expressions	supported
Link Operation Expressions	not supported
Class Extent Expressions	not supported
Sequence Construction Expressions	supported
Sequence Access Expressions	supported
Sequence Operation Expressions	not supported
Sequence Reduction Expressions	not supported
Sequence Expansion Expressions	not supported
Increment and Decrement Expressions	supported
Boolean Unary Expression	supported
BitString Unary Expressions	supported
Numeric Unary Expressions	supported
Cast Expressions	partially supported
Isolation Expressions	not supported
Binary Expressions	supported
Arithmetic Expression	supported
Shift Expressions	supported
Relational Expressions	supported
Classification Expressions	not supported
Equality Expressions	supported
Logical Expressions	supported
Conditional Logical Expressions	supported
Conditional-Test Expressions	not supported
Assignment Expressions	supported

ALF Statements	Status
Annotated Statements	not supported
In-line Statements	supported
Block Statements	supported
Empty Statements	not supported
Local Name Declaration Statements	supported
Expression Statements	supported
if Statements	supported
switch Statements	supported
while Statements	supported
do Statements	supported
for Statements	supported
break Statements	supported
return Statements	supported
accept Statements	not supported
classify Statements	not supported

Appendix D

Generated C++ Files

```
using namespace std;

/* definitions with #define */
#define INTERNALMESSAGE_NODECONN_SCIESTABLISHCONNCFM 207
#define INTERNALMESSAGE_NODECONN_SCIRELEASECONNCFM 208
#define INTERNALMESSAGE_NODECONN_RICONNECTCFM 220
#define INTERNALMESSAGE_NODECONN_RIDISCONNECTCFM 221
#define INTERNALMESSAGE_NODECONN_NODECONNECTREQ 224
#define INTERNALMESSAGE_NODECONN_NODEDISCONNECTREQ 225

/* primitive types redefinition with typedef */
typedef int CHESS_ComponentID;
typedef double Double;
typedef float Float;
typedef int Integer;
typedef string String;

/* forward declaration of types */
class CHESS_Ext_Message;
class CHESS_Int_Message;

enum NodeDisconnectResult { NODE_FORWARDED, NODE_DONE };

class NodeConn_ci_StateMachine_State;

class NodeConn_ci_State_CONNECTED;
class NodeConn_ci_State_DISCONNECTING_RESOURCE;
class NodeConn_ci_State_CONNECTING;
class NodeConn_ci_State_REQUESTING_SPAS;
class NodeConn_ci_State_DISCONNECTING_SPAS;
class NodeConn_ci_State_NODE_DISCONNECTED;
class NodeConn_ci_State_Initial;

class NodeConn_ci_StateMachine;

/* declaration of functions */
void sendInternal_I_NodeConn_PortHandler_NodeConn_riDisconnectCfm(
        CHESS_ComponentID from_ID, CHESS_ComponentID dest_ID,
        Cello_Port portId, Cello_RiServerConn serverConn_r);

class NodeConn_ci_StateMachine :
/* inheritance: */ public CHESS_StateMachine
{
    public:
    /* attributes: */
    NodeConn_ci_State_CONNECTED CONNECTED;
    NodeConn_ci_State_Initial Initial;

    CHESS_ComponentID port_NetConn_PI;

    /* methods: */
    virtual void NodeConn_riDisconnectCfm(Cello_Port portId,
                                Cello_RiServerConn serverConn_r);

};
```

Figure D.1: Portion of resulting .h file

```
#include <iostream>
#include "AAL2.h"

/* declaration of global variables: */
CHESS_StateMachine * stateMachines[16];
...
/* implementation of functions: */
void entryPoint() {

    /* local variables: */
    NodeConn_ci_StateMachine var_NodeConn_ci_StateMachine_9;
    NodeConn_ci_StateMachine var_NodeConn_ci_StateMachine_10;
    CHESS_Int_Message * msg;

    /* statements: */
    var_NodeConn_ci_StateMachine_9.startUp(&(var_NodeConn_ci_StateMachine_9));
    var_NodeConn_ci_StateMachine_9.cid = 9;
    var_NodeConn_ci_StateMachine_9.partition = "processE";
    var_NodeConn_ci_StateMachine_9.port_NetConn_RI = 7;
    var_NodeConn_ci_StateMachine_9.port_PortHandler_RI = 12;
    var_NodeConn_ci_StateMachine_9.port_SciHandler_RI = 11;

    stateMachines[9] = &(var_NodeConn_ci_StateMachine_9);
    while(1) { /* ForEver */
      msg = (CHESS_Int_Message*)(receive(sel_any));
      stateMachines[msg->componentID]->currentState = stateMachines[msg->
         componentID]->currentState->handleIntMessage(msg);
    }
}
....
void NodeConn_ci_StateMachine::NodeConn_riDisconnectCfm(Cello_Port portId,
                                    Cello_RiServerConn serverConn_r) {
    /* local variables: */
    /* statements: */
    //start inline statements (language: 'C++')
    ENTER( "NodeConn_riDisconnectCfm. " );
    //end inline statements

    if(((serverConn_r.serverConnId == serverConnId) &&
                        (connHalf[serverConn_r.connHalf].portId == portId))){
       U8 secondHalf;
       connHalf[serverConn_r.connHalf].respondState = RI_RESPONDED_CFM;
       secondHalf = NodeConn_getSecondHalf(serverConn_r.connHalf);

       if(connHalf[secondHalf].respondState){
         SciState x;
         x = ((SciHandler_ci_StateMachine*)stateMachines[port_SciHandler_RI])->
                                          SciHandler_getSciConnectionState();

         if((x == SCI_ATTACHED)){
            NodeConn_setDisconnectingSpasState();
            sendInternal_I_SciHandler_NodeConn_SciHandler_sciReleaseConnReq(cid,
                              port_SciHandler_RI, spasConnId, serverConnId);
         }
         else {
            NodeConn_setDisconnectedState();
            sendInternal_I_NetConn_NodeConn_NetConn_nodeDisconnectCfm(cid,
                                                          port_NetConn_RI);
            NodeConn_removeData();
         }}
       else {}}
    else {}
```

Figure D.2: Portion of resulting .cpp file