Model-Based Software Engineering : Methodologies for Model-Code Synchronization in Reactive System Development

van Cam Pham

université PARIS-SACLAY

list cea tech

UNIVERSITÉ PARIS SUD
Comprendre le monde, construire l'avenir

NNT : 2018SACLS611

Thèse de doctorat

Model-Based Software Engineering: Methodologies for Model-Code Synchronization in Reactive System Development

Thèse de doctorat de l'Université Paris-Saclay
préparée à l'Université Paris-Sud
au sein du CEA-LIST Saclay

Ecole doctorale n°580 Sciences et technologies de l'information et de la communication (ED STIC)
Spécialité de doctorat : Informatique

Thèse présentée et soutenue à Gif-sur-Yvette, le 12/01/2018, par

Van Cam PHAM

Composition du Jury :

Laurent PAUTET Professeur, Telecom Paristech	Président
Ileana OBER Professeur, Université Paul Sabatier	Rapporteur
Antoine BEUGNARD Professeur, IMT Atlantique Bretagne	Rapporteur
Laurence DUCHIEN Professeur, Université de Lille	Examinateur
Franck BARBIER Professeur, Université de Pau (LIUPPA)	Examinateur
Sébastien GÉRARD HDR, Chef du laboratoire LISE (CEA)	Directeur de thèse
Ansgar RADERMACHER Ingénieur-chercheur, LISE (CEA)	Encadrant
Shuai LI Ingénieur-chercheur, LISE (CEA)	Encadrant

Dedication

To

My family.

Acknowledgments

My journey to the objectives of this research work has many challenges that could not be achieved without the help of many people to whom I am indebted.

To my supervisors, Dr. Ansgar RADERMACHER, Dr. Sébastien GÉRARD and Dr. Shuai LI, I thank you from the bottom of my heart for your time, effort and advices you all have given to me. Being worked with and supervised by you is my greatest and most notable experience ever. To Ansgar, thank you very much for your thoughtfulness, distinguished and insightful technical knowledge and your detailed feedback about the thesis starting from the problem definition to the solution implementations and evaluations. To Sébastien who drives the laboratory **LISE**, please accept my gratitude to you for your motivation, patience, immense knowledge support and invaluable suggestions for the research subject that I have been following. To Shuai, many thanks to you for being a discussant and for giving me a lot of constructive comments; also, I appreciate your pedagogical capability that inspires me a lot.

To the Papyrus Designer team of the laboratory LISE, thank you all for helping me to resolve issues related to technical aspects during implementation. I am proud of being part of the best team I have ever had.

To the members developing the Moka engine, thank you very much for many discussions and for welcoming all of my questions that make me clarify the precise semantics of UML elements.

To all members of the laboratory LISE, thank you for creating such a friendly, productive and resourceful environment

To my friends, thank you very much for spending time talking to me and giving lots of encouragements every time I feel unmotivated.

Last, but not least, I would like to dedicate my sincere gratitude to the most important people in my life: my family. To my wife Van Anh and my newborn daughter Thao My, I love you so much and thank you for stepping into my life, always being by my side and making me happy all the time. To Dad, Mom and my younger brother, words are not emotional enough to express my thankfulness dedicating to you for your spiritual support and believing in me and your unconditional help; without you, I would not be able to keep my mind consistent in light of achieving the research.

Abbreviations

ALF Action Language for Foundational UML. xii, 38, 122, 134, 135
API Application Programming Interface. 23, 38
AST Abstract Syntax Tree. 25
ATL Atlas Model Transformation Language. 27

DSML Domain-Specific Modeling Language. 20, 24, 25

EMF Eclipse Modeling Framework. 21, 24, 27

FSML Framework-Specific Modeling Language. 23, 24

GPML General-Purpose Modeling Language. 9

IDE Integrated Development Environment. 2, 3, 22, 34, 38, 43, 48–50, 53, 55, 56, 61, 62, 74, 83, 99

JTL Janus Transformation Language. 28

MARTE Modeling and Analysis of Real-Time and Embedded Systems. 9
MBSE Model-Based Software Engineering. 1–5, 7–9, 14, 16, 18, 19, 26, 28, 30, 35, 38, 39, 71, 102, 113, 134

OCL Object Constraint Language. 26, 30
OMG Object Management Group. 19, 26, 61, 76, 101, 102, 118

PSCS Precise Semantics for UML Composite Structure. xii, xiii, 76, 78, 91, 97, 101, 118–122, 135
PSSM Precise Semantics for UML State Machine. xii, 103, 104, 117, 118, 120–122, 135, 141

QVT Query/View/Transformation. 4, 9, 26, 28, 29, 111

reactive system reactive system. 14, 15

TGG Triple-Graph Grammar. 4, 27, 28, 30
TML Textual Modeling Language. 83

UML Unified Modeling Language. 2, 9, 10, 12, 14, 15, 20, 22, 23, 33–35, 37, 41, 42, 44, 45, 70, 75, 76, 80, 85, 89, 91, 94, 102, 104, 106, 108, 110, 113, 117, 125, 134
UML-Class UML class. 2, 3, 20, 23, 41
UML-CS UML composite structure. xvi, 2–5, 7, 9, 10, 14–20, 22–24, 31, 36, 37, 39, 41, 45, 71, 73, 74, 89, 93, 101, 106, 117–119, 121, 124, 126, 129, 131, 132, 134–136
UML-SM UML state machine. xvi, 2–5, 7, 9–12, 14–20, 22–24, 26–28, 31, 35–37, 39, 41, 45, 48, 71, 74, 75, 81–84, 89, 102–104, 111, 113, 117, 118, 121, 124, 125, 129, 131, 132, 134–136, 139, 140, 142, 144

Model-Based Software Engineering: Methodologies for Model-Code Synchronization in Reactive System Development

Abstract: Model-Based Software Engineering (MBSE) has been proposed as a promising software development methodology to overcome limitations of traditional programming-based methodology in dealing with the complexity of embedded systems. MBSE promotes the use of modeling languages for describing systems in an abstract way and provides means for automatically generating different development artifacts, e.g. code and documentation, from models. The development of a complex system often involves multiple stakeholders who use different tools to modify the development artifacts, model and code in particular in this thesis. Artifact modifications must be kept consistent: a synchronization process needs to propagate modifications made in one artifact to the other artifacts.

In this study, the problem of synchronizing Unified Modeling Language (UML)-based architecture models, specified by UML composite structure (UML-CS) and UML state machine (UML-SM) elements, and object-oriented code is presented. UML-CSs are used for describing the component-based software architecture and UML-SMs for discrete event-driven behaviors of reactive systems. The first challenge is to enable a collaboration between software architects and programmers producing model and code by using different tools. This raises the synchronization problem of concurrent artifact modifications. In fact, there is a perception gap between diagram-based languages (modeling languages) and text-based languages (programming languages). On the one hand, programmers often prefer to use the more familiar combination of a programming language and an Integrated Development Environment. On the other hand, software architects, working at higher levels of abstraction, tend to favor the use of models, and therefore prefer diagram-based languages for describing the architecture of the system. The second challenge is that there is a significant abstraction gap between the model elements and the code elements: UML-CS and UML-SM elements are at higher level of abstraction than code elements. The gap makes current synchronization approaches hard to be applied since there is no easy way to reflect modifications in code back to model.

This thesis proposes an automated synchronization approach that is composed of two main correlated contributions. To address the first challenge, a generic model-code synchronization methodological pattern is proposed. It consists of definitions of necessary functionalities and multiple processes that synchronize model and code based on several defined scenarios where the developers use different tools to modify model and code. This contribution is independent of UML-CSs and UML-SMs. The second contribution deals with the second challenge and is based on the results from the first contribution. In the second contribution, a bidirectional mapping is presented for reducing the abstraction gap between model and code. The mapping is a set of correspondences between model elements and code elements. It is used as main input of the generic model-code synchronization methodological pattern. More importantly, the usage of the mapping provides the functionalities defined in the first contribution and eases the synchronization of UML-CS and UML-SM elements and code. The approach is evaluated by means of multiple simulations and a case study.

Keywords: UML, model-based software engineering, architecture model, UML state machines, composite structures, component-based architecture, model-code synchronization, code generation, reverse engineering, software architects, programmers.

Model-Based Software Engineering: Methodologies pour la synchronisation entre modèle et code dans le développement de systèmes réactifs

Résumé: Model-Based Software Engineering (MBSE) a été proposé comme une méthodologie prometteuse de développement de logiciels pour surmonter les limites de la méthodologie traditionnelle basée sur la programmation pour faire face à la complexité des systèmes embarqués. MBSE favorise l'utilisation de langages de modélisation pour décrire les systèmes d'une manière abstraite et fournit des moyens pour générer automatiquement de différents artefacts de développement, p.ex. code et documentation, à partir de modèles. Le développement d'un système complexe implique souvent de multiples intervenants qui utilisent différents outils pour modifier les artefacts de développement, le modèle et le code en particulier dans cette thèse. Les modifications apportées aux artefacts évoquent le problème de cohérence qui nécessite un processus de synchronisation pour propager les modifications apportées dans l'un artefact aux autres artefacts.

Dans cette étude, le problème de la synchronisation des modèles d'architecture basés sur les éléments UML composite structure (UML-CS) et UML state machine (UML-SM) du langage de l'Unified Modeling Language (UML), et le code orienté objet est présenté. UML-CSs sont utilisés pour décrire l'architecture du logiciel basée sur les composants et UML-SMs pour les comportements discrets liés aux événements des systèmes réactifs. Le premier défi est de permettre une collaboration entre les architectes de logiciels et les programmeurs produisant de modèle et de code, en utilisant différents outils. Il soulève le problème de synchronisation où il existe de **modifications simultanées** des artefacts. En fait, il existe un écart de perception entre les langages à base de diagramme (langages de modélisation) et les langages textuels (langages de programmation). D'une part, les programmeurs préfèrent souvent utiliser la combinaison familière d'un langage de programmation et d'un environnement de développement intégré. D'autre part, les architectes logiciels, travaillant à des niveaux d'abstraction plus élevés, favorisent l'utilisation des modèles et préfèrent donc les langages à base de diagramme pour décrire l'architecture du système. Le deuxième défi est qu'il existe un **écart d'abstraction** significatif entre les éléments du modèle et les éléments du code: les éléments UML-CS et UML-SM sont au niveau d'abstraction plus élevé que les éléments du code. L'écart rend la difficulté pour les approches de synchronisation actuelles car il n'y a pas de façon facile de réflecter les modifications du code au modèle.

Cette thèse propose une approche automatisée de synchronisation composée de deux principales contributions corrélées. Pour aborder le premier défi, on propose un patron méthodologique générique de synchronisation entre modèle et code. Il consiste en des définitions des fonctionnalités nécessaires et plusieurs processus qui synchronisent le modèle et le code en fonction de plusieurs scénarios définis où les développeurs utilisent différents outils pour modifier le modèle et le code. Cette contribution est indépendante de UML-CSs et UML-SMs. La deuxième contribution traite du deuxième défi et est basée sur les résultats de la première contribution. Dans la deuxième contribution, un **mapping bidirectionnel** est présentée pour réduire l'écart d'abstraction entre le modèle et le code. Le mapping est un ensemble de correspondances entre les éléments de modèle et ceux de code. Il est utilisé comme entrée principale du patron méthodologique générique de synchronisation entre modèle et code. Plus important, l'utilisation du mapping fournit les fonctionnalités définies dans la première contribution et facilite la synchronisation des éléments de UML-CS et UML-SM et du code. L'approche est évaluée au moyen de multiples simulations et d'une étude de cas.

Mots-clés: UML, model-based software engineering, modèle d'architecture, UML machines à état, composite structures, architecture basée sur les composants, synchronisation entre model et code, génération de code, reverse engineering, architectes de logiciels, programmeurs.

Synthèse en français

Synthèse: Cette thèse se déroule dans le contexte de l'utilisation d'Ingénierie Dirigée par les modèles pour les systèmes embarqués. L'hypothèse de la thèse est que les développeurs d'un système logiciel embarqué réactif complexe utilisent les éléments de modélisation Composite Structure (UML-CS) et State Machine (UML-SM) du langage UML pour concevoir le système et ils utilisent différents outils pour manipuler les artefacts de développement, le modèle et le code en particulier. Le modèle de conception et le code peuvent évoluer simultanément car les différentes pratiques, à savoir la modélisation et la programmation, modifient ces artefacts. Le problème est que, lorsque le modèle de conception et le code sont modifiés, il est très difficile de synchroniser les modifications simultanées des artefacts, car il existe un écart d'abstraction entre les éléments UML-CS et UML-SM et le code. De nombreuses approches et outils ont tenté de résoudre le problème de synchronisation des artefacts dans différents domaines, tels que la synchronisation des modèles, la programmation bidirectionnelle, la synchronisation des codes de modèles, l'ingénierie aller-retour et la co-évolution de la mise en œuvre d'architecture. Cependant, ces approches ne permettent pas de résoudre le problème posé en raison de deux problèmes: (1) la gestion des modifications simultanées du modèle et du code; et (2) prendre en charge la synchronisation entre le modèle et le code également pour les éléments avec un grand espace d'abstraction, en particulier les éléments UML-CS et UML-SM (qui n'ont pas de représentation directe dans les langages de programmation orientés objet).

Dans cette thèse, un ensemble d'exigences pour une approche de synchronization est exposé et une nouvelle approche pour le problème de la synchronisation du modèle d'architecture logicielle, spécifiée à l'aide d'éléments de modélisation UML-CS et UML-SM, et du code orienté objet est développée. L'approche proposée fait le lien entre les pratiques de modélisation et de programmation, permettant ainsi une collaboration transparente entre les développeurs. L'approche prend en charge la synchronisation des éléments structurels et comportementaux du modèle et du code. La démarche s'appuie sur les contributions suivantes:

- Mappage bidirectionnel entre un langage de programmation existant et des éléments de modélisation en ajoutant des constructions de programmation supplémentaires au langage existant pour les éléments de modélisation sans représentation en code.
- Un ensemble de modèles de génération de code permettant l'exécution de l'exécutable des constructions proposées et le débogage du code utilisateur.
- Un modèle méthodologique générique de synchronisation modèle-code pour permettre la synchronisation du code étendu avec le modèle d'architecture.

L'approche est mise en œuvre au-dessus de l'outil de modélisation Papyrus Software Designer. Plusieurs expérimentes sont menées pour évaluer l'approche proposée par rapport à l'ensemble d'exigences identifiées pour examiner les approches existantes. Les mécanismes de cartographie et de synchronisation bidirectionnels sont évalués en détail au moyen de multiples simulations d'éléments de diagramme de classes UML et de C ++, d'une étude de cas de l'exécution de Papyrus-RT développée en C ++ et d'une étude de cas de l'usine de voitures Lego utilisant UML. Éléments -CS et UML-SM. Les modèles de génération de code sont évalués pour déterminer si la prise en charge de la génération de code pour les éléments

de modélisation de la machine à états est complète, la conformité sémantique du code généré (comme défini dans UML PSCS et PSSM): l'exécution du code généré pour chaque scénario de test Les suites de tests PSCS et PSSM sont affirmées; et l'efficacité du code généré en ce qui concerne les performances de traitement des événements et la consommation de mémoire: le code généré par notre modèle pour deux exemples de machines à états est rapide et nécessite une consommation de mémoire faible par rapport aux approches incluses dans le contexte de la thèse.

Mots-clés: model-based software engineering, modèle d'architecture, machines à état, systèmes embarqués, architecture basée sur les composants, synchronisation entre model et code, architectes de logiciels.

List of publications

- V. C. Pham, A. Radermacher, S. Gérard, and S. Li. Bidirectional Mapping Between Architecture and Code for Synchronization. In *ICSA 2017 - Proceedings of the 14th IEEE International Conference on Software Architecture, Gothenburg, Sweeden, 3-7 April, 2017.*

- V. C. Pham, A. Radermacher, S. Gérard, and S. Li. Complete Code Generation from UML State Machine. In *MODELSWARD 2017 - Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development, Porto, Portugal, 19-21 February, 2017.*

- V. C. Pham, A. Radermacher, S. Gérard, and S. Li. UML State Machine and Composite Structure-Based Modeling and Code Generation for Reactive Systems. *Springer Communications in Computer and Information Science, 2017. to appear.*

- V. C. Pham, S. Li, A. Radermacher, S. Gerard, and C. Mraidha. Fostering Software Architect and Programmer Collaboration. In *21st IEEE International Conference on Engineering of Complex Computer Systems, ICECCS 2016, Dubai, United Arab Emirates, November 6-8, 2016, pages 3–12.*

- V. C. Pham, A. Radermacher, and S. Gérard. From UML State Machine to Code and Back Again! In *Position Papers of the 2016 Federated Conference on Computer Science and Information Systems, FedCSIS 2016, Gdansk, Poland, September 11-14, 2016., pages 283–290.*

- V. C. Pham, A. Radermacher, S. Gérard, and F. Noyrit. Change Rule Execution Scheduling in Incremental Roundtrip Engineering Chain: From Model-to-Code and Back. In *MODELSWARD 2016 - Proceedings of the 4rd International Conference on Model-Driven Engineering and Software Development, Rome, Italy, 19-21 February, 2016., pages 225–232.*

- V. C. Pham, Ö. Gürcan, and A. Radermacher. Interaction Components Between Components based on a Middleware. In *1st International Workshop on Model-Driven Engineering for Component-based Software Systems (ModComp'14), 2014.*

List of Figures

List of Tables

Contents

Chapter 1

Introduction

Contents

Abstract: This chapter highlights the importance and scientific challenges of the synchronization between model and code in the context of Model-Based Software Engineering (MBSE). The focus is on model elements used for reactive system development. While synchronization is important in general and has received lots of research contributions, this chapter examines particular issues of synchronization between design model for a reactive system and code. The problem and research question are then defined. Next, the assumptions for this research work are presented. Then, we provide a brief description of our contributions. Finally, we describe the structure of the thesis.

1.1 Context

Software programming is one of the most important activities during software development life cycle because of the increasing use of software for various purposes. Software-based systems now play a very important role in many domains.

The integration of more and more functions into software contributes to the increasing complexity of software-based systems, especially embedded systems. The latter are often constrained by resource infrastructure and timing requirements. An embedded system responds to stimuli from its running environment or from other systems. A reactive architecture is useful for designing flexible, loosely-coupled and scalable embedded systems [Dunkels 2006, Reactive Manifesto]. The development of software for reactive embedded systems has presented many challenges [Posse 2015]. The latter need appropriate software development methodologies to deal with. MBSE has been proposed and considered as a promising approach to address the complexity of such systems. In MBSE, a software system is represented in terms of abstract models, which provide different views of the system to stakeholders. MBSE has several advantages such as complexity management, model-based system analysis and automation [Selic 2012]. Generally speaking, this latter is the ability to automatically produce different artifacts such as documentation and code from models to raise software productivity and reduce software bugs.

Despite the many advantages of MBSE, there is, however, still significant reticence to adopt a fully model-centric approach [Hutchinson 2014, Selic 2012] in industrial practice. Perhaps there are several reasons that cause this reticence, such as the sharp distinction

between modeling and programming habits, the freedom of using text-based Integrated Development Environment (IDE)s for programming compared to the very strict structure of modeling tools, the ability to describe software behaviors in programming and modeling languages, and the tooling such port such as IDEs and version control system. In fact, there is a perception gap [Brown 2015] between diagram-based languages and text-based languages. On the one hand, programmers often prefer to use the more familiar combination of a programming language and an IDE. On the other hand, software architects, working at higher levels of abstraction, tend to favor the use of models, and therefore prefer graphical languages for describing the architecture of the system.

In order to foster the industrial adoption of MBSE, the sharp distinction between MBSE modeling and current software programming must be blurred. A process of integrating MBSE into current software companies, whose developers are very familiar with programming, should profit the advantages of both of the modeling and programming practices, where MBSE and programming practitioners can work together. Indeed, this vision has been motivated by different perspectives [Taylor 2007, Van Der Straeten 2008, Cicchetti 2016]. The collaboration between MBSE and programming practitioners producing different types of artifacts, in different languages, using different tools, raises the issue of **artifact synchronization**. In MBSE, artifact synchronization can be model synchronization, that maintains consistencies between two models, and model-code synchronization, that maintains consistencies between model and generated code.

Among many modeling languages, UML has been the most widely used modeling language in MBSE [Selic 2012, Hutchinson 2014]. UML has become an industry *de facto* standard to describe and document the architecture of complex systems [Hilliard 1999, Hutchinson 2014] despite the emergence and disappearance of a number of architecture description languages. In addition, UML class (UML-Class), UML state machine (UML-SM), and UML composite structure (UML-CS) diagrams and their visual representations prove to well capture the architecture design of a component-based reactive system [Posse 2015, Ciccozzi 2014]. Based on these observations, the focus of this thesis is to propose a model-code synchronization, where the design model is specified using UML-CS elements for describing component-based software structure and UML-SM for components' behavior.

In fact, the survey described in [Hutchinson 2014] polled stakeholders in companies who adopt MBSE approaches. The survey reveals that models are highly used for MBSE activities including the use of models for understanding a problem (95.2%), for team communication (92.7%), for capturing design (90.6%) and for code generation (88.2%). In addition, it notes that 70% of the respondents primarily work with models, but still require manually-written code to be integrated. Besides, 35% of the respondents answered that they spend a lot of time and effort merging the manually-written code to the model for avoiding the loss of this code during code regeneration from model.

In the following section, we describe our research goal and challenges.

1.2 Research challenges

Software architecture design is done at the model level by means of UML-CS and UML-SM elements and fine-grained behavior is done at the code level. Both design model and code can evolve concurrently. The problem is stated as follow:

When both design model and code are modified, it is hard to synchronize the concurrent modifications of the artifacts because there is an abstraction gap between UML-CS and UML-SM elements, and code.

Works that are strongly related to this problem include *model synchronization* and *model-code synchronization*. However, current model synchronization approaches only work under assumptions that there is an explicit traceability model between two models [Giese 2009] (not model and code) or do not consider the abstraction gap that exists between the architecture design model and code. In addition, many model-code synchronization approaches only allow either model or code to be changed at a time [Van Paesschen 2005], do not propagate code modifications to model (especially partial round-trip engineering or specialized comment approaches) [Kelly 2007], or do not take the UML-CS and UML-SM elements into account since there is an abstraction gap between these elements and code [Antkiewicz 2006]. The detailed discussion of these approaches will be presented in Chapter 2.

The problem is divided into the following research challenges.

Research challenge 1 (RC1) - Model-code synchronization methodology *How to synchronize the modifications of model and code while allowing MBSE and programming practitioners to work with their favorite artifact (model or code)?* In traditional considerations, either model or code is modified and the other artifact is updated accordingly, or modifications are made in a restricted way through unfamiliar editors provided by tool vendors or separated regions. In this thesis, we address the case where both model and code can be edited concurrently by using favorite tools of model practitioners and programmers, respectively.

Research challenge 2 (RC2) - Model-implementation abstraction gap Synchronizing the design model specified by using the UML-CS and UML-SM elements is a challenging task because there is a significant abstraction gap between the model elements and code elements [Zheng 2012]. Current artifact synchronizations are not applicable because of this abstraction gap. The synchronization can be realized if a bidirectional mapping between the model and code is established.

In the next section, we propose assumptions used in this thesis.

1.3 Assumption

We intend to solve the problem under the following assumptions:

- **Artifiact modification**: MBSE adopters (e.g. software architects) can work in a model-centric way for model manipulations while traditional programmers can continue using their favorite programming environment.
- **Fine-grained behavior code**: Model can hold fine-grained behavior code (e.g. code bodies for operations or state machine actions) as blocks of texts embedded into it, e.g. *UML opaque behavior*. This assumption does not break existing practices with UML-based modeling tools that allow to use such capability.
- **Information containment**: Model contains more information than its associated code. The code is a view of the model because not all of the information in the model

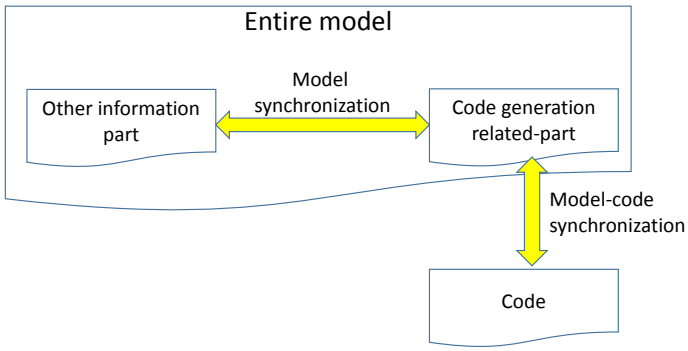


Figure 1.1: Model partitioning and information containment

is used for code generation and fine-grained behavior code can be embedded into model as the above assumption. Fig. 1.1 depicts the information contained by the model. This latter contains two model parts: *Code generation-related part* and *Other information part* such as information used for model-based security or performance analysis. The code is generated from the *Code generation-related part*. The two model parts are synchronized with each other by a model synchronization. That is to say one of the two parts might be updated correspondingly to the modifications made in the other part. Such model synchronization between the two parts is out of the scope of the thesis. Hence, we only deal with the synchronization of the code with the code generation-related model part. When we say model, we mean the code generation-related part. Except some cases we explicitly mention the two model parts. The semantics of *Code generation-related part* is similar to the concept of *skeleton* in [Seifert 2011]. This latter considers the problem of model synchronization where a model is partitioned into a *skeleton* part and a *clothing* part. The *skeleton* part is information that is shared between the two models to be synchronized and thus used during synchronization.

1.4 Contributions

The outcome of this research is to provide *an approach for automated synchronization between code and architecture model, specified by the UML-CS and UML-SM elements*. More importantly, the approach targets collaboration between software architects and programmers and allows them to work on their preferred artifact, model for the architects and code for the programmers in particular, by using their preferred language and tool.

In light of achieving this goal, we propose the following contributions for addressing the identified research challenges.

Thesis contribution 1 : In order to overcome the research challenge 1, a model-code synchronization methodological pattern is proposed. This contribution consists of definitions of necessary functionalities and multiple processes that synchronize model and code based on several defined scenarios where the developers use different tools to modify model and code concurrently. This contribution is independent of UML-CSs and UML-SMs and is detailed in Chapter 4.

Thesis contribution 2 : The second contribution is proposed based on the results from the first contribution to deal with the challenge 2. In the second contribution, a bidirectional mapping is presented for reducing the abstraction gap between model and code. The mapping is a set of correspondences between model elements and code elements. It is used as main input of the generic model-code synchronization methodological pattern. More importantly, the usage of the mapping provides the functionalities defined in the first contribution and eases the synchronization of UML-CS and UML-SM elements and code. The overview of the contributions is described in Chapter 3 and their details are presented in Chapters 4 and 5.

1.5 Thesis outline

The remaining structure of this thesis is as follows:

- Chapter 2: This chapter provides foundations of the basic concepts of MBSE as well as the UML-CS and UML-SM elements. It then presents a set of requirements that are identified as criteria for examining existing approaches and for validating the proposed approach and contributions.
- Chapter 3: This chapter presents the overview of the approach for solving the identified research challenges. Based on the approach, it describes where the contributions are positioned within the approach in order to provide readers a sketch of the research contributions. In other words, the chapter will answer the question: how are the contributions collaborated with each other to solve the problem.
- Chapter 4: This chapters describes the model-code synchronization methodological pattern in the first contribution where concurrent modifications of the model and the code are synchronized. This contribution addresses the RC1 research challenge.
- Chapter 5: This chapter disseminates the second contribution that is an approach for bidirectionally mapping between architecture model specified by the UML-CS and UML-SM elements, and code. The mapping is used as input for the synchronization methodological pattern of the first contribution presented in Chapter 4.
- Chapter 6: This chapter concludes the thesis and discusses some perspectives related to the thesis.

Chapter 2

Foundations and State of the Art

Contents

Abstract: In this chapter, we first give an overview of key concepts used as foundations in this thesis. Specifically, the basis of Model-Based Software Engineering (MBSE), the component-based modeling using UML-CS concepts such as ports and connectors, and UML-SM elements are described. The requirements for the synchronization of code and the UML elements are then proposed. These requirements are used as criteria for comparing different approaches in the literature and for validating the work in this thesis. Subsequently, the state of the art of existing approaches related to the study, including *reverse engineering*, *model-code synchronization*, *model synchronization*, *viewpoint synchronization*, *bidirectional transformation*, *view update problem*, *architecture-implementation co-evolution*, *diagram-based and textual languages*, and *code generation*, is presented to show that the research challenges are not solved yet in the literature.

2.1 Foundations

2.1.1 Model-Based Software Engineering (MBSE)

MBSE is a software engineering methodology [Brambilla 2012] that focuses on creating and exploiting models. It provides views of a system to different stakeholders participating in the development of a software project. MBSE moves the focus from programming language code in traditional programming to *models* expressed by elements of modeling languages. An element in a modeling language often has a graphical representation and a semantics.

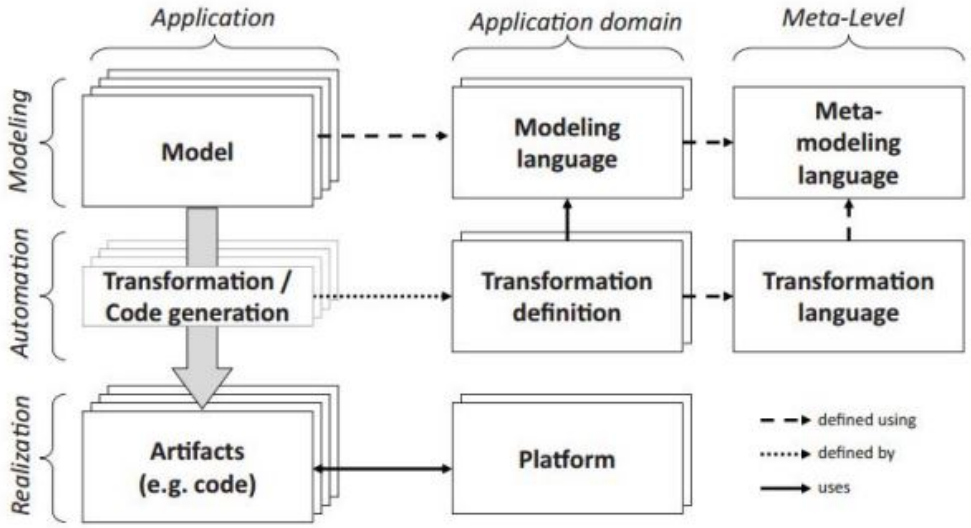


Figure 2.1: Overview of the MBSE methodology excerpted from [Brambilla 2012]

A model of a system can be used for different model-based activities such as system performance analysis or security analysis [Selic 2012]. In MBSE, a model that represents a software system can be automatically transformed into different artifacts such as code (implementation), documentation, or models conforming to another modeling language through a *transformation* or *code generation* that can be specified by specialized languages such as QVT [QVT OMG 2016] or Xtend [Bettini 2016], respectively. A transformation uses a set of rules for translating elements in one modeling language to elements in another language.

Fig. 2.1 excerpted from [Brambilla 2012] shows the overview of MBSE/MDE[1] in practice, without showing the model-based analysis such as performance analysis or security analysis. The followings give the description of the concepts appearing in Fig. 2.1.

2.1.1.1 Modeling languages

A modeling language is a mean that lets engineers specify or create models for a certain aspect of a system or a complete system [Brambilla 2012]. An element of a modeling language has its clearly defined semantics and may have graphical representations and/or textual specifications. Metamodeling is the process of defining modeling languages. The detail of metamodeling is out of the scope of this thesis.

Among many of the modeling languages, UML [Specification 2015] with its different diagrams is the most widely and extensively used for modeling software systems [Sendall 2003]. Furthermore, not just as a General-Purpose Modeling Language (GPML), UML provides means, namely *profiles*, for language engineers to extend the UML language itself to reuse a number of concepts defined in UML and express their concerns in the domain of the language engineers. An example of the UML extensions is the Modeling and Analysis of Real-Time and Embedded Systems (MARTE) language for modeling and analyzing real-time embedded systems.

[1] In this thesis, MBSE and MDE are considered as the same methodology that uses models as first artifacts to represent software/systems.

Among the different defined diagrams of UML, the UML composite structure (UML-CS) and UML state machine (UML-SM) diagrams are widely used for modeling and code generation of component-based software architecture and discrete event-driven behaviors of reactive systems [Specification 2015, Posse 2015, Ciccozzi 2014, IBM 2016a, Ringert 2014], respectively. The detailed description of these diagrams is presented in Subsection 2.1.2 and 2.1.3.

2.1.1.2 Transformation

Transformations are "*the heart and the soul*" of MBSE [Sendall 2003]. It is the process of taking as input one or more source models and producing as output one or more target models by applying a set of transformation rules. The latter can be specified by using a transformation language such as Query/View/Transformation (QVT) and established between the metalmodel elements of the source and target models.

Model-to-text transformation is a specialized transformation that takes models as input and derives text such as documentation or code as output. If code is produced, the model-to-text transformation is called *code generation*.

A transformation can be either forward or backward. A backward or reverse transformation takes as input the target models produced from the source models and reconstructs the source models as output. The process of creating an abstract model from source code is called reverse engineering.

In the next subsection, we describe the concept of synchronization.

2.1.1.3 Synchronization

Once a transformation between a source model and a target model is executed, the target model can be modified for various reasons. The modifications made in the target model must be synchronized back to the source model. Synchronization is the process of keeping multiple artifacts, that share some common information, in coherence with each other. The artifacts considered in this thesis are model and code. Synchronization can either align two models (model synchronization) or a model and its generated code (model-code synchronization). This latter is the focus of this thesis.

Modifications in the target artifact can be minor or even major. In the meantime, the source model might have changed for various reasons such as for responding to new requirements. Synchronization of these concurrently modified artifacts becomes difficult and challenging when there is a significant abstraction gap between these modified artifacts [Zheng 2012].

The next subsections describe the overview of component-based architecture modeling using UML-CS diagrams and the UML-SM elements for modeling discrete event-driven behaviors.

2.1.2 Composite structure for component-based modeling

In UML, the UML composite structure (UML-CS) diagrams provide modeling elements for describing the internal structure of a class or a component and the interaction between the internal parts of a component or between a component with other components. The elements of UML-CS are very useful in modeling a component-based software structure. The following list briefly describes the UML-CS main elements.

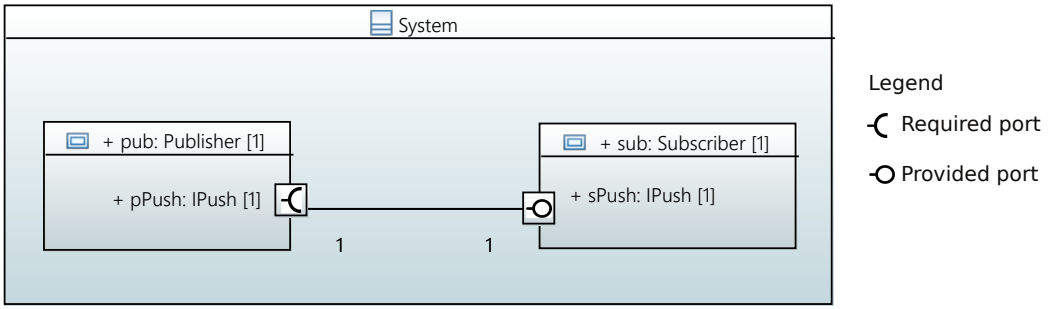


Figure 2.2: Composite structure diagram for a publisher-consumer example

- **Component**: A component can be modeled as a *UML Class* or a *UML Component*. It has its own structure composing *ports*, *parts* and *connectors*. In component-based engineering [Cai 2000], a component often serves a function of the system.
- **Part**: A part represents a role played at runtime by one or more instances of a classifier as specified by its multiplicity. A part is modeled as a composite attribute.
- **Port**: A port of a component is an interaction point between the component and its environment or between the component and its internal parts [Specification 2015]. A port can specify the services that it provides and/or the services it requires. In UML, the services are modeled as operations of interfaces meaning that a required port can request operations provided by operations of the interface of a provided port if there is a connector between them. Ports can either delegate received requests to internal parts, or they can deliver them directly to the behavior of its containing component. A port can also have a multiplicity factor.
- **Connector**: A connector specifying links enables communication between two or more instances. A connector can have multiple connector ends. Here we consider the case where a connector has only two ends. An assembly connector of a component links two or more parts or ports on parts [Ciccozzi 2016a]. A delegation connector, on the other hand, links a port p of a component to an internal part or a port on an internal part of the component. It means that what arrives at the p port will be passed on to the internal part for handling [Ciccozzi 2016a].

Fig. 2.2 shows a publisher-subscriber model example. The latter has a publisher part *pub* with a *pPush* port requiring the *IPush* interface, and a subscriber part *sub* with a *sPush* port providing the *IPush* interface. The connector linking the two ports on the two parts denotes that *pub* can send some data to *sub* via the *pPush* port by calling the operations of *IPush*.

2.1.3 UML State Machine

UML state machine (UML-SM) defines a set of concepts used for modeling discrete event-driven behavior of reactive systems. UML-SM is a significant enhanced realization of the mathematical concept of finite automaton [Rabin 1959] or finite state machines in computer science. Compared to the finite automaton, UML-SM introduces the new concepts of hierarchically nested state, orthogonal regions, state actions, transition kinds, and pseudo states. The followings describe these new concepts.

- **Hierarchical or composite state**: A composite state contains at least one region.
- **Orthogonal regions**: Regions are orthogonal to each other if they are either owned by the same state or, at the topmost level, by the same state machine. A region owns a set of vertexes and transitions, which determine the behavior flow within the region. Orthogonal regions execute concurrently if their owning state is active, meaning that multiple substates of the owning state can be active at the same time. Orthogonal regions can coordinate their execution behaviors by sending event instances to each other.
- **State actions**: A state in UML-SM can have associated actions, namely *entry*, *exit*, and *doActivity*, which are executed when the state is entered, being active, and exited, respectively. The actions give UML-SM more powerful than finite state machines in modeling complete discrete event-driven behaviors of reactive systems.
- **Transitions**: A transition in UML-SM is either *external*, *local*, or *internal*. The execution for an external transition exits the source state and enters the target state of the transition. Local and internal transitions are specialized transitions in UML-SM. A local transition only exists in composite states and its source and target vertex cannot be the same. The execution of a local transition does not exit its containing composite state. An internal transition is a local transition specializing that its source and target states are the same. The execution of an internal transition does not exit and re-enter its source/vertex state.
- **Pseudo states**: Pseudo states are used for chaining and coordinating multiple transitions. UML defines different pseudo state kinds and their associated visualizations and semantics. Pseudo state kinds are divided into five pairs: *initial/terminate*, *fork/join*, *choice/junction*, *shallow history/deep history*, and *entrypoint/exitpoint*.

Event: For modeling events, that trigger state machine transitions, UML defines four types of events: call-event, signal-event, time-event, and change-event, described as follows:

- **Call-event**: A call-event is associated with an operation/method and emitted if the operation is invoked.
- **Signal-event**: A signal-event is associated with *UML Signal* type containing data. It is emitted if the class receives an instance of the signal. When a component, whose behavior is described by a UML state machine, receives a message/signal instance through its ports, a signal-event is automatically emitted and stored in an event queue for later processing by the state machine.
- **Time-event**: A time-event specifies a wait period, starting from the time when a state with an outgoing transition triggered by the time-event is entered. The time-event is emitted if the state remains active longer than the wait period to trigger the transition. In other words, the state, which is the source vertex of a transition triggered by a time-event, will remain active for a maximal amount of time specified by the time-event.
- **Change-event**: A change-event has a boolean expression and is fired if the value of the expression changes from false to true. For example, a change-event can be used to detect changes of some information such as temperature measured by a sensor.

If no event is used for triggering a transition from a state, the transition is said: triggered implicitly by a *completion-event*. In addition, events that triggers transitions starting from pseudo states are not considered.

Fig. 2.3 shows a state machine example of a telephone, excerpted from [Specification 2015].

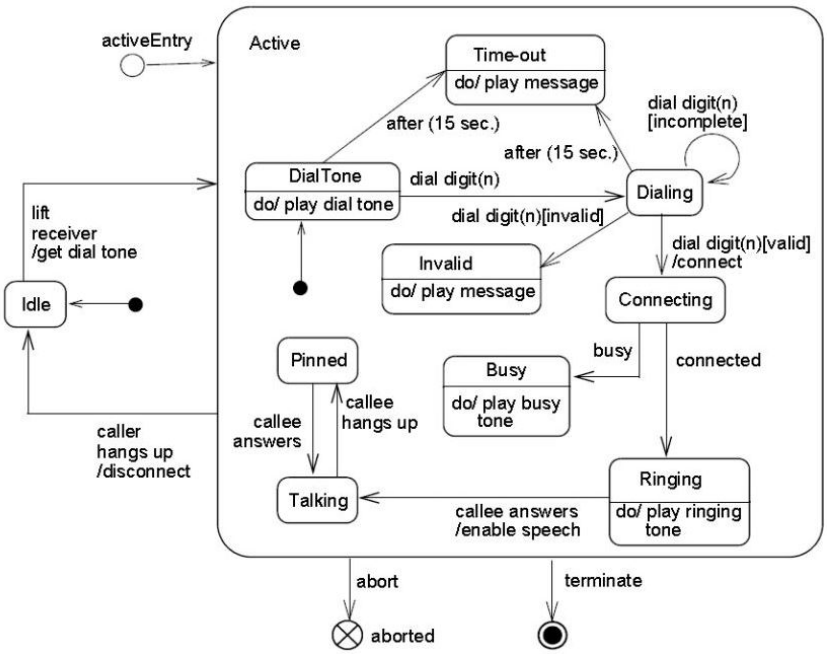


Figure 2.3: Example of a state machine excerpted from [Specification 2015]. The state machine consists of a composite state *Active*, 10 simple states, and 4 pseudo states including 2 initials, 1 terminate, and 1 entry point. Call-events such as *lift receiver* and *dial digit* are used. A time-event, which has a wait period of 15 seconds, triggers the transition from *DialTone* to *Time-out*.

2.1.4 Relations to Architecture Description ISO/IEC/IEEE 42010 standard

This subsection presents several common terminologies defined within the **ISO/IEC/IEEE 42010** *Systems and software engineering — Architecture description* international standard [ISO 2011] and relates them to this thesis. The purpose is to provide to readers a conscious understanding during reading the thesis. In fact, the ISO/IEC/IEEE 42010 standard defines many concepts related to architecture description. We only extract the concepts that are the most relevant to the thesis. The definitions of the extracted concepts are given as follows.

We first define *Concern* as follow.

Definition 1 (Concern) *A concern is any interest in a system relevant to one or more of its stakeholders [ISO 2011].*

This thesis focuses on the purpose, structure and behavior concerns of the architecture. The structure and behavior can appear at the architecture as well as implementation level,

depending stakeholders who are interested in these concerns. The concept of *stakeholder* can be defined as follow.

Definition 2 (Stakeholder) *Stakeholders are individuals, groups or organizations holding Concerns for the System of Interest. Examples of stakeholders are: client, owner, user, consumer, supplier, designer, maintainer, auditor, CEO, certification authority, architect, developer [ISO/IEC/IEEE].*

For a collaboration between stakeholders in this thesis, we focus on two specific types of stakeholders: software architects who use models to describe their concerns (structure or behavior) and software programmers who mainly work with code to realize the structure and behavior at a finer granularity.

In order to define the concept of *Architecture Model*, the definition of *Model Kind* is first given as follow.

Definition 3 (Model Kind) *A Model Kind defines the conventions for one type of modeling.*

In the terms of the ISO/IEC/IEEE 42010 standard, examples of model kinds include UML class (UML-Class) diagrams, Petri nets, UML-CS and UML-SM models. We now give the definitions of *Architecture Model*, *Architecture View*, and *Architecture Viewpoint*.

Definition 4 (Architecture Model) *An architecture model uses modeling conventions appropriate to the concerns to be addressed. These conventions are specified by the model kind governing that model.*

Definition 5 (Architecture View) *An Architecture View expresses the architecture of the system from the perspective of one or more Stakeholders to address specific Concerns, using the conventions established by its viewpoint.*

Definition 6 (Architecture Viewpoint) *An Architecture Viewpoint is a set of conventions for constructing, interpreting, using and analyzing one type of Architecture View. A viewpoint includes Model Kinds, viewpoint languages and notations, modeling methods and analytic techniques to frame a specific set of Concerns.*

Following these definitions supplied by the ISO/IEC/IEEE 42010 standard, a UML-CS diagram or a UML-SM diagram is an architecture model. In addition, an architecture view is composed of one or more architecture models. It is somewhat different from the meaning of architecture/design model used in this thesis. On the one hand, we consider that an architecture model consists of UML-CS and UML-SM model elements in this thesis. On the other hand, if we consider $UML-CS \cup UML-SM$ as a single *Model Kind*, the architecture model in this thesis and that of the ISO/IEC/IEEE 42010 standard can be considered as having the same understanding.

In the next section, requirements for synchronization of the UML elements and code are identified.

2.2 Requirements for model-code synchronization

This section identifies requirements for synchronization of the UML-CS and UML-SM elements and code for reactive system (reactive system) development. This step is important

Table 2.1: Requirements for model-code synchronization

ID	Requirement	Description
R1	Model modification propagation	If a modification in model is relevant to code, it must be reflected to code
R2	Code modification propagation	If a modification in code is relevant to model, it must be reflected to model
R3	Model modification preservation	If a modification in model is irrelevant to modifications in code, it must be preserved during propagation of the code modifications back to the model
R4	Code modification preservation	If a modification in code is irrelevant to modifications in model, it must be preserved during propagation of the model modifications to the code
R5	Concurrent modification	If there are concurrent modifications in model and code, the modifications must be synchronized to keep the model and the code consistent

because the identified requirements are used for assessing how far the support of an approach or a tool for the model-code synchronization for reactive system development is, and what is missing with the existing approaches found in the literature. Furthermore, the requirements are also a useful means for validating our contributions presented in Chapters 3, 4 and 5.

Tables 2.1 and 2.2 show our requirements for synchronization of UML-CS and UML-SM elements, and code for reactive system. The requirements are divided into two groups. The first group is related to requirements for artifact synchronization in general. The second is specific to UML-based design for reactive system development. We specify these requirements in the following definitions.

2.2.1 Requirements for artifact synchronization

Requirement 1 (Model modification propagation - R1) *Model modification propagation requires that changes made in the model, relevant to the code, must be reflected to the code by the synchronization.*

Requirement 2 (Code modification propagation - R2) *Code modification propagation requires that changes made in the code must be reflected to the model by the synchronization.*

The **R1** and **R2** requirements are derived from the *propagation* property of the model synchronization in [Xiong 2007], the *PUTGET* property in [Foster 2007], and the consistent condition of relational views in [Bancilhon 1981]. On the model side, by *relevant*, we mean that changes made to model elements that are used for code generation should be propagated to the code. Otherwise, changes irrelevant to code generation make no effect to the code. For example, changing the value of a stereotype attribute used for model-based performance analysis is irrelevant to the generated code.

Requirement 3 (Model modification preservation - R3) *Model modification preservation requires that the propagation of code modifications should preserve model elements, which are modified by some MBSE adopters and are not relevant to the code modifications.*

Table 2.2: Requirements for UML-based reactive system design

ID	Requirement	Description
R6	Structure completeness	Synchronization supports UML class and composite structure elements, especially ports and connectors of the latter
R7	Behavior completeness	Synchronization supports UML state machine elements
R8	UML runtime execution compliance (UML-conformance)	Runtime execution of code generated from UML state machines complies with the UML specification for UML state machines
R9	Generated code efficiency	The performance and memory usage of generated code

Requirement 4 (Code modification preservation - R4) *Code modification preservation requires that the propagation of model modifications should preserve code elements, which are modified by some programmers and are not relevant to the model modifications.*

The **R3** and **R4** requirements are similar the model synchronization *preservation* property in [Xiong 2007]. These requirements guarantee that modifications made in model (code) are kept intact during the propagation of modifications in code (model) to the model (code) if the modifications in the model and in the code are not in conflict.

Requirement 5 (Concurrent modification - R5) *Concurrent modification requires that if model and code are both modified, the synchronization should bidirectionally propagate the modifications between the artifacts to make them consistent again.*

2.2.2 Requirements for UML-based reactive system design

The five above requirements are similar to those of artifact synchronization in general. The following definitions give requirements specifically dedicated to the synchronization of code and model specified by UML-CS and UML-SM elements for embedded systems.

Requirement 6 (Structure completeness - R6) *Structure completeness requires that model-code synchronization must synchronize model and code if structural elements, class structure and UML-CS elements in model, are concurrently modified.*

Requirement 7 (Behavior completeness - R7) *Behavior completeness requires that model-code synchronization must synchronize model and code if behavioral elements, class operation and state machine elements, are concurrently modified in model and code.*

The **R6** and **R7** requirements allow the structure and behavior of a system can be modified both at the model and the code. These requirements give flexibility and equality between modeling and programming habits and potentially benefit from advantages of these practices.

Requirement 8 (UML runtime execution compliance - R8) *UML runtime execution compliance requires that the runtime execution of generated code conforms to the semantics defined by model.*

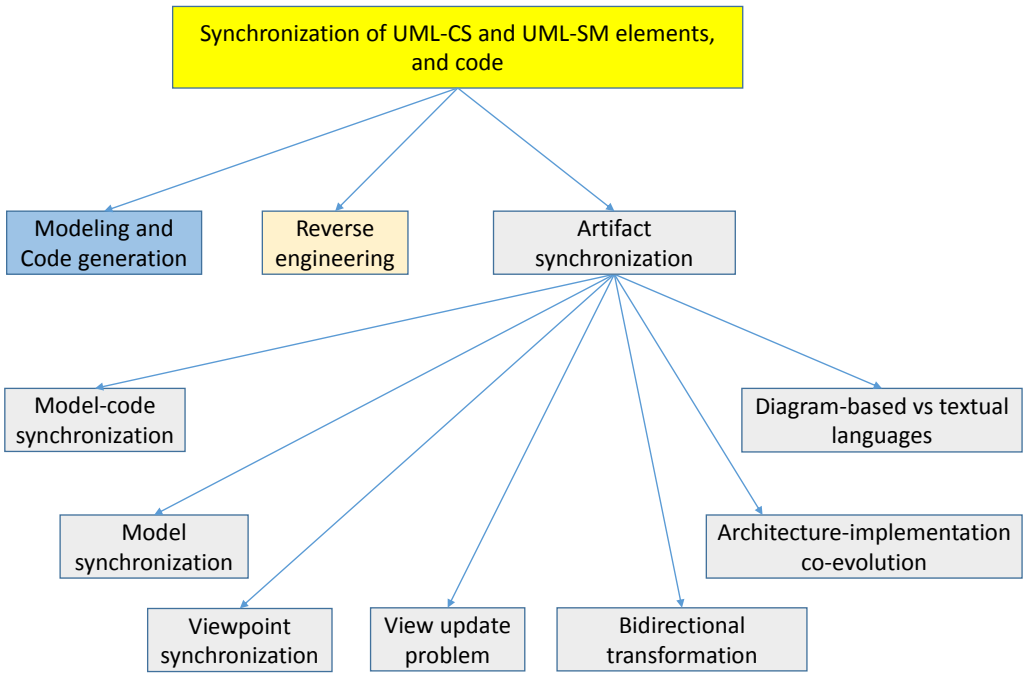


Figure 2.4: State of the art of approaches related to model-code synchronization

R8 is a must-hold requirement to assure that the generated code conforms to what is created in the model. The code must respect the semantics of structural and behavioral elements, the UML-CS and UML-SM elements in particular.

Requirement 9 (Generated code efficiency - R9) *Generated code efficiency requires that generated code must be efficient: fast in event processing and little in memory consumption.*

This thesis is to provide model-code synchronization support for embedded systems. It is then the efficiency of the code, including event processing speed in reactive system and memory consumption, must be important.

In the next section we show literature review of existing approaches for solving the problem(s) in this thesis.

2.3 State of the Art: Literature review

This section reviews different approaches proposed in the literature. The purpose of this review is to show that the research challenges identified in this thesis are not solved by the existing approaches.

The synchronization of code and UML-CS and UML-SM model elements is related to different research categories. Fig. 2.4 shows the categorization of approaches in the literature related to the problem presented in this thesis. First, the synchronization includes basic themes of software engineering, namely, forward engineering and reverse engineering. Forward engineering in MBSE is particularly composed of different steps, which create a software design model at a high abstraction level and refine the model to a detailed level

model. The latter is in turn used for code generation. We are interested in approaches for generating code from UML-CS and UML-SM elements. Code generation and reverse engineering often transform a source artifact into a target artifact. In general, these software engineering themes do not involve the propagation of modifications in the target artifact back to the source artifact. In code generation, the source artifact is model and the target artifact is code. Conversely, in reverse engineering, the source and target artifact are code and model, respectively.

Second, we consider approaches in the literature that deal with modifications in both artifacts to keep them consistent. Thus, the synchronization is related to different artifact synchronization approaches. These approaches include: model-code synchronization, model synchronization, viewpoint synchronization, bidirectional transformation languages, view-update problem in relational databases, architecture-implementation co-evolution, and diagram-based and textual languages.

The architecture-implementation co-evolution problem is considered because model in MBSE is often considered as description of software architecture. Thus, the synchronization of the model and code is a means to solve this problem.

The comparison of diagram-based and textual languages is presented because a model in MBSE is often represented in diagram-based modeling languages and code in textual programming languages. Then, a review of approaches for synchronization of diagram-based and textual languages might open possibility to solve the proposed problem.

In the followings, we first describe the review of the code generation approaches in Subsection 2.3.1. Then, the review of reverse engineering techniques is presented in Subsection 2.3.2. Subsequently, we show our review of the different approaches related to artifact synchronization in Subsection 2.3.3. Finally, we present a synthesis of the comparison of these approaches with the identified requirements in Subsection 2.3.4.

2.3.1 Modeling and code generation using UML-CS and UML-SM elements

Code generation has been extensively researched to automatically produce code from models conforming to UML or Domain-Specific Modeling Language (DSML)s. Multiple techniques are proposed for implementing code generators. However, the details of these techniques are not the focus of this thesis. We concentrate on the patterns (namely, *code generation patterns*) that are used for generating code from UML-CS and UML-SM elements and the application of these patterns to generate software.

In the followings, we first describe the code generation patterns for UML-CS and UML-SM elements. Then, we present some approaches that use these patterns to generate fully operational code (the code that is readily for compilation and execution).

2.3.1.1 Code generation patterns for UML-CS and UML-SM elements

Code generation from UML-SMs and UML-CSs has received a lot of attention in automated software development. This subsection first identifies patterns used for generating code from UML-SM elements since they receive lots of effort. We then review approaches for generating code from modeling containing both UML-CS and UML-SM elements. A systematic review of code generation proposals for UML-SMs is presented in [Domínguez 2012]. This subsection only presents the most widely used code generation patterns.

Regarding code generation for UML-SM elements, perhaps switch/if is the most intuitive

technique for implementing a "flat" state machine. Commonly, it either uses a scalar variable [Booch 1998] and a method for each event, or two variables as the active state and the incoming event used as the discriminators of an outer switch statement to select between states and an inner one/if statement, respectively. The state table approach [Douglass 1999] uses one dimension for representing states and the other one for all possible events. In fact, these approaches require a transformation from hierarchical to flat state machines. However, these approaches are hardly applied to state machines containing pseudo states such as deep history or join/fork while taking concurrent states into account.

Besides, the object-oriented state pattern [Shalyto 2006, Douglass 1999] transforms a state into a class and an event into a method. Specifically, this pattern delegates event processing from the class containing the state machine to its sub-state classes. Separation of states in classes therefore makes the code more readable and maintainable. Nevertheless, this technique only supports flat state machines. Subsequently, the authors in [Niaz 2004] extend this pattern for supporting hierarchical state machines. Recently, a double-dispatch (DD) pattern presented in [Spinke 2013] extends [Niaz 2004] to support maintainability by representing both states and events as classes, and transitions as methods. However, as the results shown in [Spinke 2013], these patterns require much memory because of an explosion of the number of classes and the use of dynamic memory allocation, which is not preferred in embedded systems. In addition, it is worth noting that none of these approaches provides implementation for all of UML-SM pseudo states as well as the four event types, namely *call-event*, *signal-event*, *time-event*, and *change-event*.

In industry, tools such as [SparxSystems 2016, IBM 2016a] apply different patterns to generate code. However, true concurrency, some pseudo-states, and UML events are not supported (see Section 5.4 on page 92 for more details). FXU [Pilitowski 2007] is the most complete tool for generating code from UML-SM elements, but generated C# code is heavily dependent on their own library. Furthermore, C# is not suitable for embedded systems. Papyrus-RT [Posse 2015] does not support concurrent states, shallow history pseudo states, and transitions from a vertex at the outside of a composite state to one of its sub-vertexes. The reason might be that Papyrus-RT transforms hierarchical state machines into flat state machines before code generation and the transformation is hardly applied to the unsupported elements.

For the purpose of generating code for different programming languages, several approaches have been proposed. Umple [Badreddin 2014] is a textual UML modeling language, which supports code generation for different languages such as C++ and Java from state machines. However, Umple does not support pseudo states such as fork, join, junction, and deep history, and local transitions. Furthermore, only call-events and time-events are specified in Umple. The ThingML [Harrand 2016] modeling language for embedded and distributed systems relies on non-UML state machines and connectors and supports code generation from these elements to various programming languages. A textual modeling language and code generation for UML are also presented in [Christian 2017].

Speaking about code generation from UML-CS elements, several tools such as Papyrus-RT [Posse 2015], IBM Rhapsody [IBM 2016a] and Enterprise Architect [SparxSysems 2016] produce code from UML ports and connectors. However, most of these tools do not consider the multiplicities of connector ends during code generation. In fact, the semantics of UML connectors for communication ports depends on the multiplicities of connector ends. The Object Management Group (OMG) standard Precise Semantics for UML Composite Structure (PSCS) precisely specifies the semantics. However, none of code generation ap-

proaches takes this standard into account. A brief description of the PSCS standard will be presented in Section 5.1 on page 66.

2.3.1.2 Application of code generation patterns

The usage of both UML-CS and UML-SM elements for reactive system modeling and code generation is commonly found in industrial tools such as Papyrus-RT and IBM Rhapsody. Nonetheless, the difference between Papyrus-RT and the targeted solution of this thesis is that the latter conforms to the UML specification and generated code does not rely on an additional runtime library, while Papyrus-RT conforms to the UML Real-Time Profile (UML-RT) [Selic 1994]. This latter consists of concepts suitable for modeling complex real-time systems [Cheng 2001]. Furthermore, code generated by Papyrus-RT relies on a runtime library - the Papyrus-RT runtime.

In order to generate fully operational code from models, we need some means to express fine-grained behavior. There are often two possible directions for it as follows:

1. Embedding fine-grained behavior code directly into model.

2. Employing a text-based model-aware action language for expressing the behavior.

Considering the first direction, industrial adoption of MBSE allows to write fine-grained behavior code into model as blocks of text. Typical tools are Papyrus-RT, Enterprise Architect [SparxSysems 2016] and IBM Rhapsody [IBM 2016a]. This practice enables integrating programming code written in common programming languages such Java and C++ into model. However, since the programming code is manually inserted, there is a possibility of producing syntactical as well as computational errors. Some tools [IBM 2016a] overcome the syntax problem by providing a code editor as a limited IDE assisting developers for inserting code. However, this support forces developers to change their favorite development practice, thus reduces more or less programming efficiency. Furthermore, when there are computational errors, developers tend to directly modify the generated code with modern IDEs with powerful support such as graphical interface-based debugging.

As for the second direction, Action Language for Foundational UML (ALF) is often considered as a potential language for expressing the fine-grained behavior at the model level in a textual way. Formally, "*ALF is a textual surface representation for UML modelling elements, whose execution semantics is given by mapping ALF's concrete syntax to the abstract syntax of the Foundational Subset For Executable UML Models (fUML)*" [Ciccozzi 2016b]. Significantly, ALF allows to preserve consistency at the modeling level. Furthermore, it enables a full-fledged code generation approach [Ciccozzi 2014]. However, what makes this latter not sound is its popularity in the programming community. It is even unknown in the world of programming with strong languages such as C++ and Java. Also, there are several questions regarding integration of ALF into the industry: how much effort do programmers spend for learning it? Does it increase or decrease productivity? How should industrial adopters integrate it? How does it interact with legacy code, e.g. Application Programming Interface (API)s of legacy code? If the syntax of ALF is much similar to Java [Ciccozzi 2016b], why should it be invented as a new language? To the best of our knowledge, these issues are not only for ALF but also for other model-aware action languages. As long as these questions are left for answering, the adoption of these action languages into industry struggles and is limited.

The review has shown the incompleteness of the existing approaches for providing code generation for all elements of UML-CS and UML-SM and the adoption of the code generation patterns for generating fully operational code from UML-CS and UML-SM elements. Thus, the requirements R6-R7 cannot be satisfied by these approaches. We argue that the production of fully operational code from models is important in MBSE to increase productivity and quality. However, since there are many developers practicing traditional programming in software development, a collaboration between the adopters of MBSE and traditional programmers should be enabled. Thus, a model-code synchronization should be realized that involves both code generation and reverse engineering.

As previously discussed, while the code generation approaches mainly concern the forward engineering of software engineering, the reverse engineering focuses on providing a model from a source code base. In the next subsection, we present our review on reverse engineering approaches.

2.3.2 Reverse engineering

Reverse engineering is commonly defined as "*the process of examining an already implemented software system in order to represent it in a different form or formalism and at a higher abstraction level*" [Chikofsky 1990, Brunelière 2014]. In practice, reverse engineering can be used in various purposes such as for updating software documentation or models from source code or finding one possible source code from artifacts such as binary files. The former is discussed in this thesis, especially approaches that can produce models from code for model understanding and manipulation in MBSE, because it is related to the **R1** requirement in Table 2.1.

Modisco Modisco [Brunelière 2014] is a Model-Driven Reverse Engineering Framework. The objectives of Modisco are to (1) discover initial models from legacy artifacts, especially source code, composing a given system; and (2) understand these models to generate relevant views on the given system. To achieve it, Modisco discovers and represents the given system as a set of models without loss of information. That is to say every code element, e.g. statement and expression, is represented in terms of model elements. The recovered system models can then be analyzed and/or re-factored using model transformations to generate other views of the system or create a new source code base of the system for deployment. For the purpose of demonstration, the authors in [Brunelière 2014] show an example that modifications in the recovered models can be propagated back to code, which satisfies the **R2** requirement. However, the recovered models seem to only provide visualizations of source code elements at a very low level, e.g. statement, operators and operands of expressions. Thus, it is not sure that the model provides better understanding than the code. Furthermore, Modisco cannot deal with the semantics of the source code such as state machine execution. Hence, the **R5-R8** requirements cannot be satisfied. JavaMoPP [Heidenreich] is similar to Modisco in the sense that everything in the code is reversed and represented as elements in model. Actually, JavaMoPP does it by providing an Ecore-based metamodel of the Java programming language so that a Java program is reversed to an instance of the metamodel.

fREX **fREX** [Bergmayr 2016] is a specific application of Modisco to discover Java code and transform the discovered Java model (based on a Java AST metamodel) to the OMG's Foundational UML (fUML) standard language. The latter is an executable format for

dynamic behavior analysis. The ATL-based transformation from the Java model to the fUML model relies on a trivial mapping from a subset of Java language elements to UML activity elements, e.g. a *MethodInvocation* in Java is mapped to a *CallOperationAction* in fUML. However, this mapping is far from obvious that there exists such clear mapping between UML-CS and UML-SM elements and Java code elements.

Software architecture recovery Architecture erosion designates the "*progressive gap observed between the planned and actual architecture of a system as implemented by source code*" [Van Gurp 2002, Terra 2012]. Software architecture recovery is considered as a software architecture repair technique used for controlling architecture erosion [De Silva 2012] and for software re-engineering. The recovery process extracts architectural information from its source code by using reflexion models [Murphy 2001], dependency and call graphs [Mancoridis 1999], abstract syntax graph of source code for matching design patterns specified as graph transformation rules [Niere 2002], or query language-based recovery [Sartipi 2003]. Recovered architectures in the approaches are, however, not the intended architectures, but are often supplemented by an *architecture reconciliation* process using refactoring techniques to obtain the intended architectures [De Silva 2012]. That means that a complete automatic process for recovering an intended architecture from its implementation is missing. Therefore, doing the process manually multiple times during development might negatively affect the development process. Furthermore, the authors in [De Silva 2012] pointed out that despite the support of a number of tools and techniques, architecture recovery does not have broad adoption in industry. Usually, UML models are extracted from source code combining with design documentation.

The process of extracting UML models from source code is supported by many commercial tools and research prototypes. Many tools [IBM 2016a, SparxSysems 2016, Magic 2016, Paradigm 2016] support the creation of UML class diagram elements from the source code. A systematic review of the tools for reverse engineering for UML-Class is found in [Cutting 2015]. Several tools such as UMLLab [Yatta Solutions 2012] and MagicDraw [Magic 2016] provide the propagation of code modifications back to model in real-time: a change in code is immediately reflected back to the model. However, the propagation is limited to some structural UML-Class elements. A few tools such as MagicDraw and Visual Paradigm [Paradigm 2016] support the construction of a sequence diagram for code statements of a method. However, if the sequence diagram and its code concurrently change, no synchronization is supported: modifications in either the diagram or the code are ignored. A few approaches [Abadi 2012, Sen 2016] support extracting state machines from sequential code by using static analysis of source code. However, the extracted state machines in these approaches are often different from the original ones and are mainly used for system understanding, rather than code generation and synchronization. None of these tools supports the synchronization of UML-CS and UML-SM elements and code as the problem that is addressed in this thesis.

2.3.3 Artifact synchronization

Differently from code generation and reverse engineering where only one of the artifacts, model and code, is modified, artifact synchronization allows to make modifications in both artifact and keep them consistent. In the followings, we present our review on the approaches related to the artifact synchronization. We start with the two categories: model-code synchronization and model synchronization because they are proposed in the context

of MBSE. Next, we elaborate the review to other research categories including viewpoint synchronization and bidirectional transformation that can be applied to the artifact synchronization in MBSE. Then, we show some approaches in the relatively classical categories: view-update problem and archiecture-implementation mapping because they also have the issue of keeping two different artifacts consistent. Finally, we discuss works related to the use of diagram-based and textual languages for modeling and some approaches that synchronize these two representations.

2.3.3.1 Model-code synchronization

Model-code synchronization has been an important aspect since the creation and use of DSMLs. The need of the model-code and model synchronization (that is described in the next subsection) has triggered a technique called *round-trip engineering* [Sendall 2003, Hettel 2008] that automatically maintains the consistency of the artifacts once they are modified. Multiple model-code round-trip engineering approaches are described in this section. Round-trip engineering can be divided into *partial* and *full* round-trip engineering. Partial round-trip engineering is a way to integrate user-code into generated code and preserve it from being overwritten by code generators. Different mechanisms for partial round-trip engineering have been proposed to separate the code generated from the model from the user-code, which is subjective to be manually modified. The followings describe the solutions proposed for partial round-trip engineering.

Protected Regions A protected region is a section of the generated code that can be modified. Generally, the developers should not modify any code outside of protected regions. Actually, the latter are separated from the generated code by using specialized/dedicated comments. For illustration, Listing 2.1 shows an example of using a protected region for the *getName* method of the *GeneratedClass* class, separated from other elements by *PROTECTED REGION* comments/tags. As a result, a modification to the code within this protected region remains as it is after code regeneration because the generator recognizes the specialized comments. Also, the Eclipse Modeling Framework (EMF) framework [Steinberg 2008] implements this approach to allow developers to insert their code in the protected regions to replace the default code generated by the framework.

Listing 2.1: Protected region example

```
class GeneratedClass {
public:
  string name;
  GeneratedClass(string name) {
    this->name = name;
  }
  // changes outside of protected regions will be *overwritten*
  public string getName() {
    // PROTECTED REGION ID(GeneratedClass.getName) ENABLED START
    // changes inside of protected regions are *protected*
    return name;
    // PROTECTED REGION END
  }
}
```

Although this approach seems quite elegant, it causes more problems than it solves. Because the protected regions require a density of comments in the code, it requires that developers are so highly disciplined that the comments are never broken. If either accidental modifications occur in the comments or modifications are outside of the protected regions, the modified code within the region is erased by code regeneration because the code generator no longer recognizes these comments [Zheng 2012] to preserve the modifi-

cations. Further problems with protected regions include modifications of the model which lead to the invalidation of manually written code (such as renaming of classes and methods) [Kelly 2007].

Fine-grained code within model To overcome the limitations of protected regions, many UML tools such as IBM Rhapsody [IBM 2016a], Enterprise Architect [SparxSysems 2016], and Papyrus-RT [Posse 2015] allow to embed fine-grained behavior code/user-code directly within model to keep a single source containing all necessary information. This practice is consistent with the **Information containment** assumption in Section 1.3. In UML, *OpaqueBehavior* is a mechanism to contain code associated with an operation. By this way, fully operational code can be generated without the need of modifying the generated code. Modifying the fine-grained code is realized right at the model level. However, the code modification practice does not allow programmers to use their favorite IDEs or programming editors, thus might reduce their efficiency and productivity.

Full round-trip engineering It points to approaches that allow changes in both model and code. Generally, this type of round-trip engineering requires that a synchronization mechanism must support both directions of change propagation: model to code and vice versa. This synchronization is non-trivial to realize in practice because it requires, at least, a bidirectional mapping between model and code. Nonetheless, this bidirectional mapping currently does not exist between the model elements in UML-CS and UML-SM. Even if there exists such a bidirectional mapping, the synchronization is still a challenge if model and code are both modified. Actually, there are several attempts to solve this full round-trip engineering problem.

Fujaba Fujaba [Klein 1999] abbreviates for "*From UML to Java and Back Again*". It is a public research prototype. It aims to support round-trip engineering for UML diagrams, both structural and (to some extent) behavioral diagrams. Software structure and behavior in Fujaba are modeled as UML class (UML-Class) diagrams and so-called *story diagrams*, respectively. Story diagrams are the combination of UML activity diagrams and collaboration diagrams. The generated code in Fujaba contains pre-defined implementation patterns, using naming conventions. Actually, the latter are combined with a certain amount of annotations in the generated code for reconstructing the software model. Fujaba can satisfy the requirements **R1** and **R2**. However, to the best of our knowledge, the requirements **R3** and **R4**, which involve the preservation of artifact modifications during change propagation, are not supported in Fujaba, especially regarding UML-SM. Furthermore, software architecture in Fujaba is not modeled in terms of UML-CS.

Tool support for round-trip engineering of UML-Class elements and code Several commercial and open-source tools [SparxSysems 2016, IBM 2016a, Magic 2016, Yatta Solutions 2012] support the round-trip engineering between UML-Class elements and code. Generally speaking, this round-trip engineering allows developers to modify the code and propagate the modifications back to the original UML model containing UML-Class elements. Systematic reviews of some of these tools are available in [Cutting 2015]. Support for Java round-trip is prominent in most tools. Other languages such as C++ are only available in a few tools [IBM 2016a, SparxSysems 2016, Magic 2016]. Importantly, these tools can satisfy the **R1-R4** requirements, but there is no methodology for supporting the other require-

ments. In addition, Enterprise Architect [SparxSysems 2016], IBM Rhapsody [IBM 2016a], and Magic Draw [Magic 2016] allow to modify generated code within protected regions and the modification is preserved during code regeneration. However, the mechanism to support this preservation feature is based on the protected region techniques, which present other problems as previously discussed.

Round-trip engineering using Framework-Specific Modeling Language (FSML) [Antkiewicz 2006] Usually, many applications for a framework rely on specific provided concepts such as supported APIs and object-oriented classes of the framework. Unfortunately, the use of the provided concepts to create an application is obviously not easy because the application developers need to know which framework-provided concepts are available and how to instantiate them correctly. In addition, the instantiation often involves multiple steps, which create multiple framework-based objects. These objects produce the desired effect if correctly created. In order to deal with the challenges during creation of framework-based applications, the authors in [Antkiewicz 2006] propose Framework-Specific Modeling Language (FSML)s. Generally speaking, the latter are a special category of DSMLs and are created by using concepts provided by the framework. An instance of a Framework-Specific Modeling Language (FSML) is then used in a *forward engineering* process for generating pieces of code using the provided concepts for interacting with the framework. The authors also state that the code generated from the FSML instance can be used in a *reverse engineering* process for reconstructing the instance model. Through the support of the forward and reverse engineering, the authors then define an *agile round-trip engineering* to make the instance model and the generated code synchronized in case there is a concurrent modification of the two artifacts. However, the proposed round-trip engineering can only deal with framework-specific applications. In fact, the reverse engineering requires the modifications in the generated code to follow very strict rules conforming to the FSML so that this process can understand and parse the modified generated code. Hence, if the rules are not respected, the round-trip engineering is broken. Furthermore, this approach relies a lot on a specific framework while UML-CS and UML-SM elements are framework-independent. Another point to remember is that a FSML model is used for generate a code segment of the application code. It means the code always contains more information than the FSML model. In contrast, in our **Information containment**, model contains more information than code.

SelfSync In [Van Paesschen 2005], the authors propose SelfSync as a dynamic round-trip engineering environment. It aims to synchronize a data modeling view and its corresponding implementation objects written in the *Self* object-oriented implementation language [Ungar 1987]. The authors assume that the data modeling view and the implementation contain the same information. For round-trip engineering, the authors propose four scenarios: (1) view entities are modified, the implementation is updated accordingly; (2) relationships between view entities are modified, the implementation is updated accordingly; (3) implementation objects are modified, the data modeling view is updated accordingly; and (4) relationships between implementation objects are modified, the data modeling view is updated accordingly. In addition, the update process is in real-time in the direction from the modeling view to the implementation, which means that modifications in the modeling view are instantly propagated to the implementation. Compared to the requirements in this thesis, SelfSync can satisfy the requirements **R1-R4** but not the other ones.

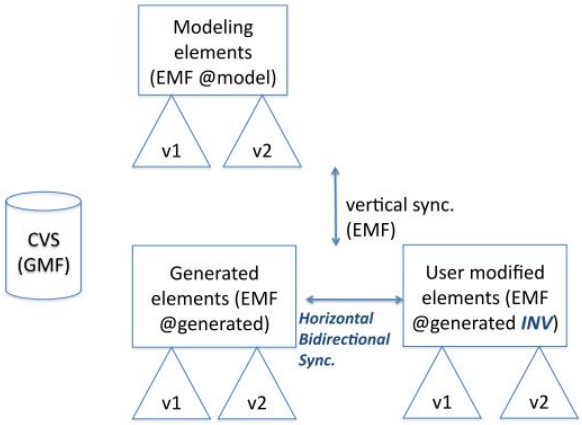


Figure 2.5: Bidirectional invariant traceability framework by horizontal bidirectional synchronization excerpted from [Yu 2012]

Maintaining invariant traceability through bidirectional transformations In [Yu 2012], the authors deal with the problem of maintaining consistency between an Ecore-based model and code generated by Eclipse Modeling Framework (EMF). Commonly, EMF generates default behavior code for Ecore-based metamodel elements. If users customize this default behavior code and preserve the customizations by using *@generated NOT* comments or users change code generation templates for generating the default behavior code, a "*mismatch problem*" between the default behavior code, namely *template-generated code*, and the user-modified code might occur. Specifically, the authors propose a framework that combines the existing vertical synchronization of model and code in EMF, which is based on protected regions as previously discussed, and a horizontal bidirectional synchronization. The overview of the approach is shown in Fig. 2.5. Actually, the horizontal bidirectional synchronization is a code-code synchronization or code merging rather than a model-code synchronization. Furthermore, a specialized comment *@generated INV* is introduced to annotate methods, that contain default behavior code. In effect, the purpose is to merge user modified code of a *@generated INV*-annotated method with code of the same method generated by the code regeneration in case model or code generation templates is changed.

Syntactic Model-Code Round-Trip Engineering The authors in [Angyal 2008] propose a round-trip engineering approach for fully and syntactically synchronizing model and code. The goal is to synchronize model conforming to a DSML and code to provide iterative development. In the latter, the model and the code can be concurrently modified and the proposed round-trip engineering synchronizes the modifications during development. They propose to synchronize the code with a platform specific model, which is very similar to the Abstract Syntax Tree (AST) of the code, using a three-way approach. The platform-specific model is then synchronized with the model conforming to the DSML. However, despite claiming support for model-code round-trip engineering, the proposed approach is only the synchronization of the AST in the code space with an AST model conforming to an AST metamodel in the model space. This approach then assumes that the synchronization of the AST model in the model space can be synchronized with the original model conforming to the DSML by using model synchronization approaches. This assumption is too strong and non-trivial because the hard point of the synchronization of model and code

is the abstraction gap between the obtained AST model and the DSML model.

Model-code consistency Model-code consistency checking helps detect inconsistencies between the design model and source code during their co-evolution. The authors in [Riedl-Ehrenleitner 2014] argue that model and code in MBSE do evolve frequently and concurrently, which is consistent with our argument. Therefore, there is a need to check the consistency between the evolving artifacts. They propose an incremental consistency checking approach that detects inconsistencies between the model and the code instantly. Their framework integrates both UML model and Java code into an in-memory common representation. This latter is used by a consistency checker that relies on a set of consistency rules written by developers in constraint languages such as Object Constraint Language (OCL). The checker incrementally detects and informs the developers about a project's consistency status during development. However, they do not intend to reconcile consistencies by a synchronization mechanism if inconsistencies are detected.

2.3.3.2 Model synchronization

As MBSE is increasingly used in industry, model synchronization is indispensable in the daily usage of MBSE. Model synchronization aims to maintain consistency between a source model (or a set of source models) and a target model (or a set of target models). Indeed, this consistency problem raises when two models, sharing some information, are modified. Many approaches are proposed in MBSE for model synchronization. Actually, model synchronization works are usually categorized by their model transformation which can be total, injective, bi-directional, or partial non-injective [Hettel 2008].

Query/View/Transformation (QVT) The QVT OMG standard [QVT OMG 2016] consists of three languages for model transformation. Among them, QVT-Relations (QVT-R) is a declarative bidirectional model transformation language. As the QVT specification says, it supports pattern matching and creates traces to record what have occurred during a transformation execution. Generally, a bidirectional transformation in QVT-R is specified as a set of relations between elements of a source model and those of a target model. A single specification of relations is used for bidirectional transformation in both directions: from source to target model and vice versa. From the created specification, modifications in one model can be propagated to the other model. In fact, two case studies of the applicability of QVT-R are shown in [Westfechtel 2015] and [Greiner 2016]. In the latter, even though the authors claim round-trip engineering UML class models and Java source code, using bidirectional transformations with QVT-R, what they do is synchronizing the UML models with the Java model of the Java code. This is because QVT-R cannot deal with synchronization for text-based representation of code. One might argue that code can be transformed into an AST model, e.g. the Java AST model of Java code, and this AST model and be synchronized with UML models as in [Greiner 2016, Westfechtel 2015]. However, as previously discussed, the question is *how do we map the AST model to UML-CS and UML-CS elements that have an abstraction gap*? Regarding the backward transformation of QVT-R, as revealed by the authors in [Cicchetti 2010], some tools supporting QVT-R such as Medini [Kiegeland 2014] produce undesired results: in Medini, when reconstructing the source model from a target model without modifications, the reconstructed source model differs from the original one. Furthermore, the work in [Stevens 2010] points out several semantic issues of QVT-R.

Triple-Graph Grammar (TGG) Triple-Graph Grammar (TGG) [Giese 2009] is a model transformation and synchronization technique based on graph theory. In TGG, a transformation is specified as a graph-based specification. In general, mapping rules between elements of the source and the target metamodel are graphically defined. A rule consists of a source metamodel element (or a pattern involving a set of source elements), namely the left hand side of the rule, and a target metamodel element (or a pattern involving a set of target elements), namely the right hand side of the rule. From the graph-based mapping specification, Java code is generated to execute the actual transformation. Commonly, both forward and backward directions of the transformation can be generated to support model synchronization. In [Hermann 2012], the authors formalize TGG for synchronizing concurrently modified models. TGG has shown its applicability and feasibility in several case studies in [Giese 2010] in which the authors use TGG to keep SysML [Holt 2008] and AUTOSAR [Fürst 2009] models consistent. However, the synchronization approach in [Giese 2010] requires that only one of the models is changed at a time. Furthermore, TGG for model synchronization requires an explicit and typed traceability model that maintains the links between the left hand side and the right hand side. In fact, the traceability model must be persisted with the source and the target model in the same model store so that the traceability model can refer to the elements of the source and the target model, using XMI identifiers of model elements [Bergmann 2012]. Therefore, in applying TGG to the model-code synchronization, the main issue is, however, how to create this traceability given that code elements have no associated XMI identifiers. In addition, other issues are related to the complexity of the graph-based specification between UML-SM elements and code elements: the code is usually too fine-grained, thus hard to represent by a graph specification.

Atlas Model Transformation Language (ATL) : Atlas Model Transformation Language (ATL) [Jouault 2006] is a model transformation language running on EMF. Transformations written in ATL are compiled to ATL byte code, which is executed in the ATL Virtual Machine. In fact, ATL originally does not support backward propagation: modifications in the target model of an ATL model transformation are not propagated back to the source model. Seeing that, the authors in [Xiong 2007] propose to automatically derive *put-back functions* associated with the model elements of the source model of an ATL transformation. Furthermore, they extend the semantics of each ATL byte-code instruction, which also associates the generated values with appropriate putting back functions. When a model element in the target model is modified, appropriate putting-back functions are called to propagate modifications back to the source model. In addition, a synchronization mechanism based on differencing source and target model states to detect modifications is then proposed. However, addition of an element in the target model that can be reflected to the source model is not well handled according to the authors. A "toy case study" of synchronization of UML classes and XMI-based representation of Java is presented to evaluate the approach. Nevertheless, the approach is too ATL byte code-specific and the case study is too simple where an obvious mapping between model and code can be provided. Hence, it is not clear how it can be applied to more complex cases such as UML-SM and code that have an abstraction gap.

Change-driven model transformation Change-driven transformation [Bergmann 2012] aims to incrementally detect changes made in a source model and propagate the changes

to a target model using pre-defined transformation rules. Unlike TGG and QVT-R, model synchronization with this approach must be done through two unidirectional transformations. Therefore, this approach is dedicated to model transformation, especially incremental model transformation, that updates the target model following changes in the source model, rather than model synchronization.

Janus Transformation Language (JTL) : In [Cicchetti 2010], the authors propose JTL as a declarative model transformation language supporting non-bijective transformation and change propagation. Generally, *non-bijective* transformation is the ability of transforming a model into a set of other models, e.g. a target model might semantically correspond to a family of source models. In their work, JTL has a QVT-R-like syntax and uses the Answer Set Programming (ASP) language and the DLV solver [Leone 2006] to execute transformation and change propagation in both directions: forward from source to target and backward from target to source. In this approach, the backward propagation of changes in a target model creates a set of source models, instead of one. However, little changes in a target model usually correspond to a combinatorial explosion of the source model space [Eramo 2015]. In [Eramo 2015], the authors present an approach based on JTL for managing the source model space returned from little changes in the target model. The authors demonstrate a small change in the target model of a transformation from a hierarchical state machine to a flat state machine, the back propagation of this small change results in 48 different source models. Consequently, a change in code generated from a hierarchical state machine would correspond to many source models, which are hard for programmers to accept the synchronization. Therefore, they propose to use *uncertainty models* to manage the many generated source models. However, they require to change the existing metamodel of the source model for adding elements for uncertainty management. Furthermore, the approach relies on *trace* elements each of which refers to a single source model element and a single target model element. These trace elements are impossible for model-code synchronization since a model element might be translated into multiple code elements (declarations/definitions/statements/expressions) at multiple places in the source code repository. In our approach that will be presented in Chapters 3, 4 and 5, code changes related to UML-SM elements are unambiguously propagated to only one UML-SM.

In the literature, a few approaches provide formal definitions for round-trip engineering and artifact synchronization as in [Hettel 2008]. The authors of the latter propose an abductive logic programming-based approach [Hettel 2009] which propagates changes from one model to the other model to achieve the synchronization. However, it is not clear how this approach can handle the concurrent modification problem.

As a final note, model synchronization has been extensively studied in MBSE. The majority of the proposed approaches can satisfy some of identified synchronization requirements, **R1** and **R4** in particular, related to the propagation of model modifications to code. Nevertheless, it is not clear how these approaches can deal with the requirements R5-R9, especially the case when there is a significant abstraction gap between model and code that are both modified.

2.3.3.3 Viewpoint synchronization

Generally, viewpoint modeling is one of the ways to cope with the complexity of a complex system by dividing the design phase, according to several areas of concerns [Eramo 2008]. Viewpoints enable the partitioning of the model of a system into several representations

[Finkelstein 1991]. Each viewpoint usually has its own language for expressing its particular concern. Elements in each viewpoint are related to elements in other viewpoints to ensure the consistency and completeness of the system globally. Ordinarily, the relationships between the elements of viewpoints are expressed in terms of correspondences [Eramo 2008]. To keep the viewpoints consistent, changes made in the representation of a viewpoint need to be propagated to representations of other viewpoints. Both model and code can be considered simply as different views of the same system. Informally, the ability of propagating changes between viewpoint representations is viewpoint synchronization. This latter is considered as model synchronization where the representation of a viewpoint plays the role of a model.

In [Grundy 1998], the authors focus on the management of inconsistencies between artifacts used during analysis, design and implementation of multiple-view and multiple-user integrated tools. As a result, they propose a generic architecture with tooling support that supports the representation of the specifications and the detection, definition, representation, and propagation of inconsistencies between the specifications. However, according to [Eramo 2008], they do not have any well-defined underlying architectural framework that allows the precise and explicit specification of viewpoints, views, and correspondences between them.

For the purpose of improving the modeling of relationships and constraints between elements in different viewpoints, the authors in [Eramo 2008] propose an approach for better guaranteeing the consistency. The approach is based on a change management mechanism that consists of three steps: *change identification*, *change classification*, and *change commitment and propagation*. The steps are very similar to those of incremental model transformation [Kusel 2013]. Subsequently, the authors in [Romero 2009, Ruiz-gonzález 2009] extended this approach for a mechanism of managing correspondences between the elements of different viewpoints. In their approach, they combine *extensional* and *intensional* correspondences by using QVT. Generally speaking, *extensional* correspondences are established at the instance level of the viewpoints while *intensional correspondences* are at the level of the metamodels of the viewpoints.

In [Kramer 2013, Kramer 2015a], the authors propose a generative approach for viewpoint synchronization. The approach is based on the Orthographic Software Modeling (OSM) concept [Atkinson 2008] where a single model is maintained and views are generated from this single model. Therefore, the synchronization between views becomes a propagation of changes made in one view to the single model and another change propagation from the latter to other views. Indeed, the significance of this approach is that it automatically derives the synchronization mechanism from a set of declarative correspondence rules, consistency invariants, and imperative actions written in a domain-specific language and OCL. Hence, developers do not need to deal with technical details of the synchronization.

In [Goedicke 2000], the authors argue that inconsistencies will exist in systems developed by different actors, using different viewpoints. Being that, they suggest that tools must be able to tolerate inconsistencies. As a result, they propose a distributed graph transformation to deal with the problem of formalizing the integration of multiple viewpoints in software development. However, their work focuses on requirements engineering, while this thesis targets specifically both model and code.

In contrast to the above approaches that mainly focus on the synchronization of views of viewpoints represented as models, this thesis targets specifically both model and code.

Viewpoint synchronization usually does not consider code because code is deemed to be too fine-grained. Thus, modeling of explicit relationships between model and code elements is non-trivial. Furthermore, there is a significant abstraction gap between model and code that hinders the modeling of the relationships.

2.3.3.4 Bidirectional transformation languages

A *bidirectional transformation (bx)* written in a bx programming language is a mechanism for maintaining the consistency of two related sources of information. Bx can be applied to various areas such as Model-Based Software Engineering (MBSE) and relational databases [Hu 2011, Czarnecki 2009]. A bx program is either symmetric [Diskin 2011a, Hofmann 2011] or asymmetric [Diskin 2011b]. In asymmetric bx, one source of information, e.g. code, is a view of the other source, e.g. model. It means that a source of information contains more information that the other source. This characteristic is identical to the **Information containment** assumption presented in Section 1.3 on page 3. In contrast, in symmetric bx, each source of information contains information that cannot be represented in the other source. Bidirectional transformation languages can be applied to many areas of artifact synchronization from basic data structures such as strings [Barbosa 2010, Foster 2008] and lists [Hofmann 2011] to complex data such as software models [Diskin 2008]. However, "*as the two transformations always update one artifact according to the other, bidirectional transformations do not allow parallel updates on the two artifacts*" [Xiong 2009]. The following paragraphs review approaches for asymmetric bx since it is related to the model-code synchronization with the assumptions in this thesis.

Lens techniques [Foster 2007] are originally proposed for tree synchronization. Generally, a lens consists of a *get* function, which computes the view v (the source of information that contains less information that the original source), and a *put* function, which updates the source s if the view is updated. The formalisms of these two functions are as follows:

$$get(s) = v \tag{2.1}$$
$$put(s, v') = s' \tag{2.2}$$

where s' and v' are the updated source and view. A lens-based bidirectional transformation is considered reasonable if the following conditions are satisfied:

$$put(s, get(s)) = s \tag{2.3}$$
$$get(put(s, v')) = get(s') = v' \tag{2.4}$$

Based on the lens formalisms, various languages are proposed for writing lenses to execute bidirectional transformations for basic data formats such as lists and complex data structures such as graphs.

In order to write lenses in practice, various approaches have been proposed. In [Wider 2011], the authors embed lens functions into Scala to do model transformations bidirectionally. Boomerang [Bohannon 2008] allows to write lenses for bidirectional transformations between string data. In addition, Boomerang also deals with the problem of order in collections [Hidaka 2011]. Subsequently, matching lenses [Barbosa 2010] generalize Boomerang by lifting the update translation strategy to support a set of different alignment heuristics. Besides, the authors in [Matsuda 2015] propose to use functional programming to deal

with bidirectional transformations. In [Hidaka 2010], the authors challenge the problem of bidirectional graph transformations by providing a formal definition of a well-behaved bidirectional semantics for UnCAL, which is a graph algebra form the graph query language UnQL [Buneman 2000]. Their approach is also tested for several examples. However, the tested examples usually have a trivial bidirectional relationship or mapping between the elements of the source information and those of the target information. Unfortunately, this bidirectional mapping is hardly to find in the case of UML-CS and UML-SM elements, and code. Thus, there is still a big gap in the application of bidirectional transformations to the synchronization of model and code when there is a significant abstraction gap.

In summary, bidirectional programming languages are useful for transformations between basic data structure formats such as list, string or more complicated such as XML and models. However, there is still a gap in applying them to synchronize the high level design model and the source code. This is because bx programs mainly maintain the consistency of two data formats with a non-significant abstraction gap, especially the semantics gap between the model and the code addressed in this thesis. Furthermore, in case of model and code, a model element is transformed into (many) lines of code that are located in multiple separate locations, while the examples demonstrated by bx programs operate on simple data, e.g. a string in a data format is mapped to another string contained by a pair of tags in an XML file.

2.3.3.5 View update problem

Commonly, code can be considered as a view of the system generated from the design model. Code is then similar to the concept of views in relational databases. Views in a relational database provide different representations for the database. A view is created by queries written in Structured Query Language (SQL), e.g. a view allows to show the name and the date of birth of a person while the database can contain other information such as address and phone number.

In practice, a view is changeable and the changes should be updated back to the database. However, the propagation of the changes from a view to its underlying database is not trivial, which is known as view-update problem [Chen 2011, Hettel 2010]. The challenge is that given a view created by a query, there is often not a unique way for translating an update made in the view to the database. Sometimes, there is no way for translation and sometimes there are more ways. To deal with it, many authors have proposed various approaches since 1980s, such as [Bancilhon 1981] (see the next paragraph for more details). Regarding to the classification of the proposed approaches, the authors in [Chen 2011] present a survey of the approaches for the view-update problem.

The authors in [Bancilhon 1981] address the view-update problem with the concept of *view complement*. This latter is defined as the information of an underlying database that is not exposed (or not visible) in a view when this view is created by using queries. Additionally, the authors argue that a view complement and the corresponding view are sufficient to recompute the whole database. In their proposed approach, given a *constant complement* of the view, there is only one way to put view updates back to the underlying database [Xiong 2009] and the complement remains invariant. The authors in [Dayal 1982] argue that it is not always possible to translate a view update to the underlying database. They then derive conditions under which the translation of view updates will produce correct results. Additionally, other approaches in those years focus on analyzing the characteristics of views [Masunaga 1984] and view-update policies [Medeiros 1986], and designing translation

mechanisms [Masunaga 1984].

Besides, the view-update problem is also studied in the context of bidirectional transformation languages presented in Subsection 2.3.3.4. Several approaches based on lenses [Foster 2007, Barbosa 2010] have been recently proposed in this context. However, these approaches do not directly address the complexity of the view-update problem since these approaches only deal with simple data structure such as trees [Foster 2007] and strings [Barbosa 2010].

2.3.3.6 Architecture-implementation mapping and co-evolution

Maintaining the consistency between architecture and implementation is crucial in current software development. Unfortunately, both architecture and implementation might evolve frequently, thus introducing inconsistencies between the modified architecture and the modified implementation. Researchers have proposed many approaches for addressing this problem. The proposed approaches in the literature range from conformance checking, architecture recovery as discussed in Subsection 2.3.2 on page 20 and architecture-implementation co-evolution. In fact, reverse engineering techniques presented in Subsection 2.3.2 for architecture recovery can be considered as potential hints for the consistency problem. However, there are differences between the latter and reverse engineering. Ordinarily, consistency maintenance involves keeping both architecture and code (implementation) consistent. Therefore, the starting point of the consistency maintenance requires both architecture and implementation, while reverse engineering approaches start from a source code base.

Informally, architecture conformance checking verifies whether an implementation complies to constraints defined in its architecture. Usually, architecture conformance checking can be considered as a step of detecting inconsistencies during the co-evolution of architecture and implementation. However, the conformance checking is out of scope since our focus is how to propagate changes made in the architecture to the implementation and vice versa during co-evolution. In other words, given that there are inconsistencies between the two artifacts, how to make them consistent again.

In the sequels, we review several current approaches for architecture-implementation co-evolution.

ArchJava ArchJava in [Aldrich 2002] unifies a formal architecture specification and its implementation in a single entity [De Silva 2012], allows flexible implementation techniques, ensures traceability between architecture and code, and supports the co-evolution of architecture and implementation [Aldrich 2002]. Furthermore, ArchJava guarantees *communication integrity* between an architecture and its implementation, even in the presence of advanced architectural features like runtime component creation and connection. With this intention, ArchJava extends Java by adding new language constructs to it. Specifically, the new language constructs are *component*s, *connection*s, and *port*s, which are available in UML. Thus, the constructs help programmers to describe architecture right in the implementation and enable a synchronization of model and code. In addition, the authors provide an ArchJava compiler for generating Java standard code from the code written in ArchJava. However, ArchJava does not conform to UML. First, a port in ArchJava can provide and/or require methods instead of interfaces as in UML. Second, *connections* in ArchJava do not support multiplicities for connector ends for designing different connection patterns. Third, code in ArchJava cannot be directly used for execution and debugging since only

```
[List 2]
01: interface component cSubject {
02:  pointcut getState():execution(String getState());
03:  pointcut setState():execution(void setState(String));
04:  pointcut notify():execution(void notify());
05:  pointcut notifyObservers() :
06:    cflow(execution(void setState(String)))
07:      && call(void notify());
08:
```

Figure 2.6: Example of Archface excerpted from [Ubayashi 2010]

the ArchJava compiler can understand the language syntax and semantics and no debugger is available for ArchJava. Thus, ArchJava is only Java backward compatible, thus not efficiently used with programming assistance in existing Java IDEs [Abi-Antoun 2005]. Last, behavioral elements are not supported in ArchJava. Consequently, we consider ArchJava as an architecture description language (ADL) that allows to embed Java code within a single entity rather than a programming language that can be executed and debugged by modern IDEs and standard compilers.

Archface Archface in [Ubayashi 2010] tries to solve the problem of co-evolution of design and code by proposing an interface mechanism. This latter in fact plays a role as ADL at the design phase and programming interface at the implementation phase [Ubayashi 2010]. Similarly to ADLs, architectural design in Archface is based on the component-and-connector architecture. In detail, Archface exploits technologies of Aspect-Oriented Programming (AOP), such as pointcut and advice, to specify the collaboration among components of the architecture. There are multiple Archface pointcuts such as *class*, *method* and *field* for designing structure, and *call*, *execution* and *cflow* for designing behavior. Furthermore, architecture constraints in Archface are based on these pointcuts. For instance, Fig. 2.6 shows multiple constraints specified using Archface in the design. Specifically, the constraints for implementing *cSubject* are: (1) three public methods *getState*, *setState*, and *notify* must be defined; and (2) *notify* must be called under the control flow. Based on the design in Archface, a programmer can *manually implement* the system. During the evolution, the Archface compiler detects errors or inconsistencies if the implementation does not conform to the design [Ubayashi 2012]. Thus, either the design or the implementation should be manually modified accordingly to the other artifact. However, in Archface, the implementation is not automatically generated and the Archface compiler only detects inconsistencies between the design and implementation rather than makes them consistent again. Furthermore, Archface relies on AOP that makes its implementation limited to aspect-oriented programs.

1.x-mapping In [Zheng 2012], the authors propose an approach, namely *1.x-mapping*, for tackling the problem of architecture-implementation evolution of Java code and model written in the *xADL* [Dashofy 2001] architecture description language. They argue that existing approaches for architecture-implementation consistency management are found inefficient in (1) mapping code changes back to architecture; (2) mapping architecture changes to code; and (3) supporting for behavioral mapping. To deal with these issues, 1.x-mapping puts architecture-prescribed code (or architecture-generated code) and user-code in different programming constructs, classes in particular, instead of separating user-code from architecture-prescribed code by using specialized comments. In addition, 1.x-mapping relies

on a semi-incremental code regeneration and architecture change monitoring and management. In particular, 1.x-mapping records architecture changes and propagates them to the corresponding implementation by only regenerating architecture-prescribed code. For the purpose of supporting behavior, 1.x-mapping uses a subset of UML sequence diagram-like and UML-SM elements (only simple states, external transitions and effects are supported) to model architecture behaviors. However, this approach has several issues: (1) 1.x-mapping only allows to modify the architecture at the model level; (2) programmers are not able to change the design at the code level; (3) developers must write detailed statements such as local variables within sequence diagrams by using the modeling tool to change the sequence diagram-based behavior of a component; and (4) an architecture of components in 1.x-mapping does not conform to UML and only a small subset of sequence diagram and UML-SM elements is utilized.

2.3.3.7 Diagram-based and textual languages

Generally, diagram-based languages, also called graphical languages, are a subset of visual languages. In fact, a number of efforts were undertaken that compared textual languages and visual languages.

In [Schanzer 2015], the authors discuss the pros and cons of visual languages, compared to textual languages, in various contexts. Others [Brown 2015] discuss what limits the adoption of visual languages.

Contrary to these works, the objectives of this thesis do not strictly oppose textual languages to visual languages like diagram-based languages. Instead, the approach that we seek prefers to blur the boundaries between them. Actually, the idea that visual languages can co-exist with textual languages has been noted by other authors. Besides, others suggest that the gap between visual languages and textual languages is not as huge as it is perceived by both the MBSE and software engineering communities.

In [Conversy 2014], the authors argue that the gap between textual and visual languages is narrow. As a result, they propose a framework to represent code in a visual language to improve comprehension of the code.

In domains such as requirements engineering, tools, such as IBM Rational Doors [IBM 2016b] and Visure Requirements [Visure 2016], are conscious of the importance of supporting requirements written traditionally in plain text by developers. Notably, these tools allow the transformation of requirements written in a textual human language to use-cases and structured requirements.

Generally, one of the artifact is only used to assist comprehension. There is no need for synchronization between artifacts because the transformation only proceeds in one direction.

In order for developers to be more efficient, they should not be forced to choose between a diagram-based or textual language. Works [Von Mayrhauser 1995] that emphasize the role of software comprehension, for efficient software maintenance and evolution, date back to as far as the late eighties/early nineties. In this thesis, we consider specifically both software architects and programmers. The former generally foster diagram-based languages for architecture description and comprehension. On the other hand, the latter prefer textual languages since they are deemed much more expressive for specifying fine-grained algorithms, for example.

Some approaches allow to do modeling in a textual way. Representatives of these approaches include the works in [Addazi 2017, Jouault 2014, Engelen 2010, Grönniger 2014, Maro 2015]. Most of these works focus on providing the ability of manipulating modeling

Table 2.3: Existing approaches with respect to the model-code synchronization requirements. R1 = Model modification propagation; R2 = Code modification propagation; R3 = Model modification preservation; R4 = Code modification preservation; R5 = Concurrent modification. "N/A" = not applicable; "+" = support; "-" = not; "+-" = partial (either require manual intervention or produce unintended results or need significant effort if used).

Requirement	*Code generation*	*Reverse engineering*	*Model-code sync*	*Model sync*	*Viewpoint sync*	*Bidirectional*	*View-update*	*Architecture-implementation*	*diagram- vs text-based*
R1	+	N/A	+	+-	+-	+	+-	+	+-
R2	N/A	+-	+	+-	-	+-	-	+	-
R3	N/A	+-	+-	+-	+-	+-	N/A	+-	N/A
R4	+-	N/A	+	+-	-	+-	N/A	+-	N/A
R5	N/A	N/A	-	-	-	-	N/A	+-	N/A

elements, especially UML model elements, in a textual editor. The sets of supported UML elements include the model elements of UML activity, UML-Class, UML-CS, UML-SM and some stereotypes. These approaches rely on Xtext [Bettini 2016] with facilities for creating textual languages. In fact, the textual modeling offers several advantages over graphical modeling such as productivity, version control, and textual modifications at a fine granularity [Grönniger 2014]. These advantages are consistent with our argument that allowing to work in both graphical and textual representations has many advantages. Some of the approaches [Addazi 2017, Maro 2015] provide synchronization of the textual and graphical representations. However, the synchronization only allows to modify one of the two representations at a time. More importantly, what these approaches try to do is not model-code synchronization since the textual representation of the model is not code. A textual modeling language is not a mainstream programming language with multiple IDEs that assist the development by integrating many other tools such as compilation and debugger. Furthermore, a programming language also has a richer set of language features such as expressions and statements than a textual modeling language.

2.3.4 Synthesis

This subsection synthesizes the results of comparing what the previously reviewed categories offer with the identified requirements. Tables 2.3 and 2.4 show the synthesis for the two sets of requirements.

Regarding the model-code synchronization requirements, each of the reviewed categories provide some of the requirements. Code generation approaches can provide model modification propagation (R1) and code modification preservation (R4) since these requirements are related to forward engineering. Reverse engineering approaches support limited capa-

Table 2.4: Existing approaches with respect to the requirements for UML-based design. R6 = Structure completeness; R7 = Behavior completeness; R8 = UML-conformance; R9 = Generated code efficiency; "N/A" = not applicable; "+" = support; "-" = not; "+-" = partial (either require manual intervention or produce unintended results or need significant effort if used).

Requirement	*Code generation*	*Reverse engineering*	*Model-code sync*	*Model sync*	*Viewpoint sync*	*Bidirectional*	*View-update*	*Architecture-implementation*	*diagram- vs text-based*
R6	-	N/A	-	-	-	N/A	N/A	+-	N/A
R7	-	N/A	-	-	-	N/A	N/A	-	N/A
R8	+-	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
R9	+	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A

bilities for code modification propagation (R2) and model modification preservation (R3) since these requirements related to creating model from code. Most of approaches in the categories model-code synchronization, model synchronization, bidirectional transformation and architecture-implementation provide a limited level of satisfaction for the R1-R4 requirements. For example, some of the model synchronization approaches require translating the AST of Java code to a Java model in the model space and synchronizing the Java model with a UML model [Greiner 2016]. That means the UML model is synchronized with the Java model that is persistently stored, but not the Java code. Furthermore, they do not offer any means to synchronize model-code in case of concurrent modification. Some of the architecture-implementation approaches provide hints to satisfy the R5 requirement but they often recover architecture models that are not the intended results (not the same as the original architecture models) (see Subsection 2.3.3.6 on page 32 for more details).

Speaking about the UML-based design requirements, a few of the reviewed approaches are applicable. Most of the approaches related to architecture-implementation co-evolution offers the R6 requirement but not the others because they often deal with structural aspects of software architectures only. In addition, most industrial UML code generation tools such as IBM Rhapsody [IBM 2016a] and Enterprise Architect do not provide code generation for all of the UML-CS and UML-SM concepts (see Section 5.3 on page 80 for more details).

2.4 Summary

This chapter presented the foundations and the state-of-the art in the context of UML composite structure (UML-CS) and UML state machine (UML-SM) elements and model-code synchronization. We started by revisiting the basic concepts of MBSE including *modeling languages*, *transformation*, and *synchronization*. We then briefly described the semantics of the UML-CS and UML-SM elements which are important in the context of

the thesis. Next, we identified a set of requirements based on the literature that the proposed approach should satisfy. The purpose of the requirements is to serve as partial criteria for comparing the existing approaches with the objectives of the thesis. Furthermore, they are also used for evaluating and validating the contributions of the thesis. Finally, we showed the review of the existing approaches as well as the research topics related to the problem being addressed by thesis. In fact, we also used the requirements identified in this chapter to explore what are missing in these approaches.

In the next chapter, we show the overview of our approach to the proposed problem and our contributions during solving the problem.

Chapter 3

Overview of approach and contributions

This chapter describes the overview of the approach proposed in this thesis and its contributions for addressing the research challenges identified in Chapter 1.

Our approach is inspired by the research challenge proposed by Woods in [Woods 2010]. Woods argues that current practices have a lack of architectural information in the implementation. This lack leads to a series of the problems related to software architecture such as outdated architecture description or undesired recovered architecture from code. Subsequently, the author proposes to integrate explicit architectural information into the implementation. Eventually, a number of benefits of this integration are discussed: (1) keeping implementation aligned with architecture; (2) recovering an intended architecture from its implementation; and (3) reducing the drift between architecture and implementation.

Returning to the context of this thesis, we think that the synchronization of an architecture model, specified by UML-CS and UML-SM elements, and code can be possibly realized if architectural elements can be explicitly represented in an implementation language. With this intention, we raise the abstraction level of an existing programming language by extending it with architectural modeling elements that have no representations in this programming language. We describe the overview of the realization of the approach in the followings.

Fig. 3.1 shows the overview of the approach. The latter consists of a mapping mechanism between architecture model and code, and a model-code synchronization mechanism. This synchronization uses the mapping as a means to synchronize the model and code in case of concurrent modifications. The mapping mechanism is formed by a bidirectional mapping and an in-place text-to-text transformation that acts as a preprocessing. The followings describe the concepts in the figure.

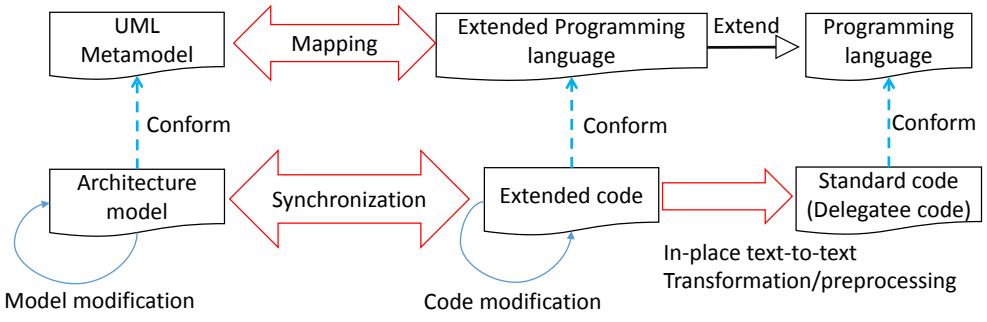


Figure 3.1: Overview of approach and contributions

UML Metamodel The subset of the UML metamodel is used for modeling and designing component-based design of UML-based reactive systems. The subset essentially consists of the concepts supported by UML class (UML-Class) and UML composite structure (UML-CS) diagram elements for structure design (see Appendix A for more details), and UML state machine (UML-SM) elements for discrete event-driven behavior design.

Architecture model The architecture designed by using the concepts of the subset of the UML metamodel. The architecture model might be modified by the software architects or indirectly updated by the synchronization, which propagates modifications in code back to the model.

Programming language It is a (standard) programming language, especially an object-oriented programming language such as C++ or Java, that is intended to be synchronized with the UML-based **Architecture model**.

Standard code It is the code conforming to the **Programming language**.

Extended Programming language As discussed in Chapter 1, one of the identified research challenges is the abstraction gap between the model and the standard code. The motivation behind this question is that modeling elements are at a higher level of abstraction than that of standard code elements [De Silva 2012]. To bridge the gap, our idea is to raise the abstraction level of the standard programming language. We extend the latter by adding new programming constructs for modeling elements that have no direct representations in common programming languages, notably UML ports, connectors, and state machine elements. We choose these model elements because ports and connectors are widely used in many Architecture Description Languages (ADLs), and state machines are suitable to model the discrete event-driven behavior of components in reactive systems. The **Extended Programming language** is the standard language with the additional constructs. It is as close as possible to the existing standard programming language in order to be suitable for programmers perception to minimize additional learning efforts. The additional constructs are created by using specialized built-in features or mechanisms of the standard programming language such as templates, and macros in C++ or annotations in Java. The use of the **Extended Programming language** term is used for distinguishing the language, which has the additional constructs, from the standard language. The details of the constructs are presented in Chapter 5. At this point, it is at best to know that because the constructs are syntactically valid to the standard code, the extended code is valid to the standard code.

Extended code Code that conforms to the extended programming language. It uses the additional constructs, that are added to the standard programming language, to express related architectural information right at the code. The extended code syntactically conforms to the standard programming language because the additional constructs are created by the built-in features of the standard programming language. By this way, the extended code can seamlessly reuse legacy code or library written in the standard programming language. This is especially important because current development of complex systems relies a lot on library support [Zhai 2016]. On the other hand, programming facilities such as syntax highlights and auto-completion in IDEs for assisting the development are reused.

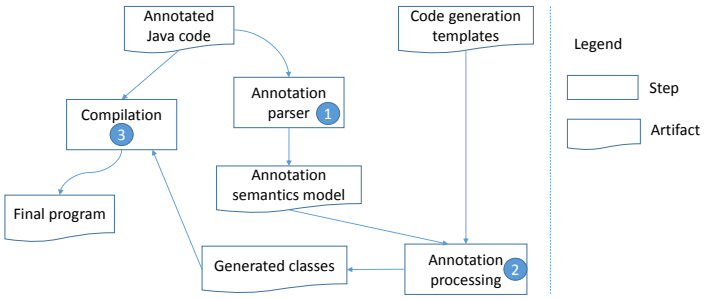


Figure 3.2: Annotation processing in Java

Programmers, working with the extend code, can therefore change the architectural information or the fine-grained code behavior at the code level while the code modifications can be reflected to the model by the synchronization mechanism.

Mapping The mapping consists of a set of correspondences [Brambilla 2012] between the modeling elements of the UML metamodel subset and the extended language. This mapping is bidirectional and is used as a means to ease the model-code synchronization.

In the following sections, we use the terms *Extended code* and *code* interchangeably and *Extended language* refers to the standard language with our additional constructs.

In-place transformation/Preprocessor The additional programming constructs in the extended programming language are compilable by standard compilers such as GCC (because the constructs are syntactically valid to the programming language), and ease the connections from code to architecture model. However, they are natively not executable. The in-place transformation indeed acts as a preprocessing step. Let's explain why the extended code itself is not executable and why there is a need to have the transformation. The extended code and the transformation act exactly similar to annotated Java code and a Java annotation processing [Pawlak 2016] in case of Java at compile time.

In Java, annotations are means used for embedding metadata in program code. They are used as markers by frameworks for altering the behavior of the annotated program code. Fig. 3.2 shows how the annotations are used [Pawlak 2016]. The alteration is realized by generating additional source code based on a set of code generation templates and the semantics of the annotations [Deors 2011]. In Step 1, the annotated Java source code is parsed for extracting a semantics model of the annotations. In Step 2, the annotation processing takes the semantics model and the code generation templates to produce a set of newly generated classes. Lastly, the final program is composed of the annotated Java code and the generated classes in Step 3.

Projecting the extended code and the transformation in the thesis into the Java annotation processing, the extended code, model semantics and (in-place) text-to-text (T2T) transformation correspond to the annotated Java code, the annotation semantics model and the annotation processing, respectively. Similar to the Java annotation processing that generates additional classes, namely *Generated classes* in Fig. 3.2, at compile time, the T2T transformation also produces a set of classes from the additional constructs of the extended code. We call the classes generated by the transformation *delegatee* classes and the standard code containing these delegatee classes *delegatee code*. The final executable code

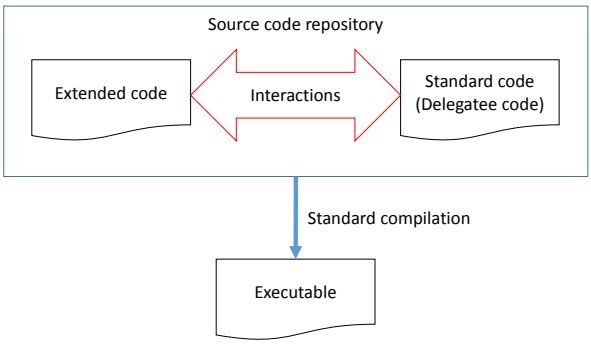


Figure 3.3: Extended code and Standard code in the same repository

is then composed of the extended code and the generated delegatee code. The delegatee code can be defined as follow.

Definition 7 (Delegatee code) *Delegatee code is hidden from developer's perspectives and consists of multiple standard source code files or classes that are generated from the extended code by the T2T transformation. It is part of the final code used for compilation and execution.*

Note that, even though Fig. 3.1 conceptually separates the extended code and the standard code generated by the transformation, namely *delegatee* code as previously described, these code entities are physically located within the same repository as shown in Fig. 3.3. There are execution interactions between these code entities that will be detailed in Chapter 5. At this point, it is at best to know that the delegatee code generated from the extended code exists as an additional code part, and a standard compiler such as GCC must take as input both of the extended code and the generated standard code to produce an executable.

Model modification Software architects or modeling users make modifications/changes to the architecture design model during development.

Code modification Programmers make modifications/changes to the extended code during development.

Synchronization Once the model and/or the code are/is modified, the synchronization mechanism reflects modifications made in one artifact to the other artifact and vice-versa (see Section 5.5 for more details).

Discussion The idea of adding new programming constructs for architectural elements to an existing programming language is similar to ArchJava [Aldrich 2002]. However, as discussed in Subsection 2.3, there are multiple fundamental differences between ArchJava and the approach presented in this thesis. First, ArchJava does not conform to UML and a port in ArchJava can provide and/or require methods instead of interfaces as in UML. Secondly, a major difference between the two approaches is that code in ArchJava cannot be directly used for execution and debugging while the extended code and the complementing standard code generated by the T2T transformation designed in this thesis are completely compilable by standard compilers, thus executable and debug-able. ArchJava does not have

this ability since ArchJava is only Java backward compatible and developers using ArchJava must debug the code generated from the ArchJava code instead [Abi-Antoun 2005]. Therefore, if there are bugs in the code generated from ArchJava, programmers cannot directly modify the generated code but manually relate the code segment containing the bug to the ArchJava code segment. Thus, ArchJava is simply an architecture description language rather than a programming language that can be executed in standard IDEs. In our approach, the extended language is truly executable and programmers modify it directly if they find bugs. Our support is more practical since programmers usually prefer to work with a programming language that they can modify and debug directly. Lastly, behavioral elements are not supported in ArchJava. Furthermore, connectors represented in ArchJava do not support multiplicities for connector ends for designing connector patterns between components that will be detailed in Chapter 5. Therefore, our approach has a broader scope than ArchJava.

Several approaches [Christensen 2011, Krahn 2006] use annotations to embed architectural information into Java. The authors in [Christensen 2011] discuss the need to have architectural information embedded within source code in agile development methods where "*the code is the central artifact*". They propose to use annotations to describe design patterns directly within the source code. They argue that valuable architectural information is retained by parsing the annotations. This approach is code-centric. The architectural information embedded in code describes the design patterns and the role of each code element in the design patterns rather than relationships with UML elements at the model level while we consider both model and code as development artifacts. The use of Java annotations is also discussed in [Krahn 2006] where annotations for *Part* and *Port* are proposed. These annotation-based approaches are not generative. It means that the annotations are not used for generating other code as the text-to-text transformation in our approach does (see Sections 5.3 and 5.4 for more details). In contrast, the annotations are only used for describing or documenting the code.

Contributions The approach above that will be detailed later in this thesis contains the contributions described as below.

Thesis Contribution 1 (TC1) - Synchronization This contribution is to provide a model-code synchronization mechanism for concurrent modifications of model and code. It addresses the research challenge 1 (RC1). Specifically, we propose a model-code synchronization methodological pattern. This latter defines and explores different processes dedicated for software architects and programmers to synchronize model and code, using their preferred tool of choice. The description and formalization of this contribution are detailed in Chapter 4.

Thesis Contribution 2 (TC2) - Mapping This contribution addresses the research challenge 2 (RC2). It provides a bidirectional mapping between the extended code and the UML metamodel subset including UML-CS and UML-SM elements. This mapping is used as input for the synchronization mechanism in Thesis Contribution 1 to address the synchronization problem between UML-CS and UML-SM elements, and code. The final result is that architecture model specified by the UML elements and the extended code can be synchronized in case there is a concurrent modification of the two artifacts. In addition, to make the extended code executable and debug-able as previously described,

an in-place text-to-text transformation is proposed. This transformation relies on several code generation patterns. The bidirectional mapping and the transformation are presented in Chapter 5.

Chapter 4

A model-code synchronization methodological pattern

Contents

Abstract: This chapter presents a methodological pattern for model-code synchronization supported by corresponding tooling. The methodological pattern addresses the Thesis Contribution 1 for solving the research challenge **RC1**. The contribution proposes to use automated model-code synchronization as a means of bridging the gap between model edited by software architects, and code written by programmers. Our approach is based on the following sub-contributions, described in detail later in this chapter:

1. A generic model-code synchronization methodological pattern to solve the problem of collaboration between different types of developers, when model and code co-evolve. This contribution is based on the following requirement and proposition:
   - Requirement: the availability of a generic IDE with functionalities necessary for model-code synchronization. The functionalities are not dependent of a particular approach or technology. The required IDE will be described in Section 4.1.
   - Proposition: Processes to use the IDE to synchronize model and code based on several defined scenarios, which correspond to common development practices (see Section 4.2).
2. An Eclipse-based implementation of the approach for synchronization between UML models and corresponding C++ code (see Section 4.3).

3. Experimentation-based evaluations of the model-code synchronization approach to the artifact synchronization requirements in Table 2.1 and applying the proposed solution for the development of a real-world example (see Section 4.4).

This contribution is presented before the bidirectional mapping because it is a generic methodological pattern. It can be applied to a specific model-code synchronization, e.g. synchronization of UML-SM elements and code that will be presented in Chapter 5, whenever the functionalities of the IDE are available.

Contrary to traditional solutions, we generalize our approach by considering the case that the model is not only used for architectural design, but also for full implementation since model can contain fine-grained behavior code as stated by the assumptions in Section 1.3.

The rest of the chapter is organized as follows. Section 4.1 presents the IDE, different actors that use the IDE, and the availability of generic functionalities in the IDE. The processes for synchronization are then presented in Section 4.2. Section 4.3 instantiates an implementation of the approach. Evaluations of the approach are presented in Section 4.4. We conclude the chapter in Section 4.5.

4.1 Collaborating actors and use-cases of synchronization

In this section we define the actors who will use our model-code synchronization approach to collaborate during development. Then we define the main functionalities, as use-cases, expected from a generic IDE used by these actors. Some basic concepts related to the actors and use-cases are also defined in this section.

4.1.1 Collaborating actors and development artifacts

For the sake of generality, we postulate that the architect and programmer are actors with starkly opposite development practices. This allows the approach to be used even in cases where model and code can both be used for the full implementation of a system, rather than just architectural design for the former, and code implementation for the latter. First, we introduce the concepts of *development artifact* and *baseline artifact*.

Definition 8 (Development artifact) *A development artifact is an artifact, as defined in [OMG 2008], that can be used for the full implementation of the system.*

For example a system can be entirely implemented as code. The code is then a development artifact. Similarly, a model may also be a development artifact. It is then not only a documentation of the specification but also is used for generating the implementation. For example a model can be used for implementation by generating code from the model, and compiling the code without the need to edit or complete the code. As previously discussed in Section 2.3.1 on page 17, full implementation of a system can be generated from the model by embedding user-written code as blocks of text associated with model elements, e.g. UML *OpaqueBehavior*. However, this embedding reduces programming efficiency and thus is not preferred by programmers who prefer using modern IDEs with well-supported development assistance such as syntax highlights and auto-completions. Model and code are both development artifacts in a model-code synchronization mechanism. A development artifact may be the baseline artifact, defined in this thesis as follows:

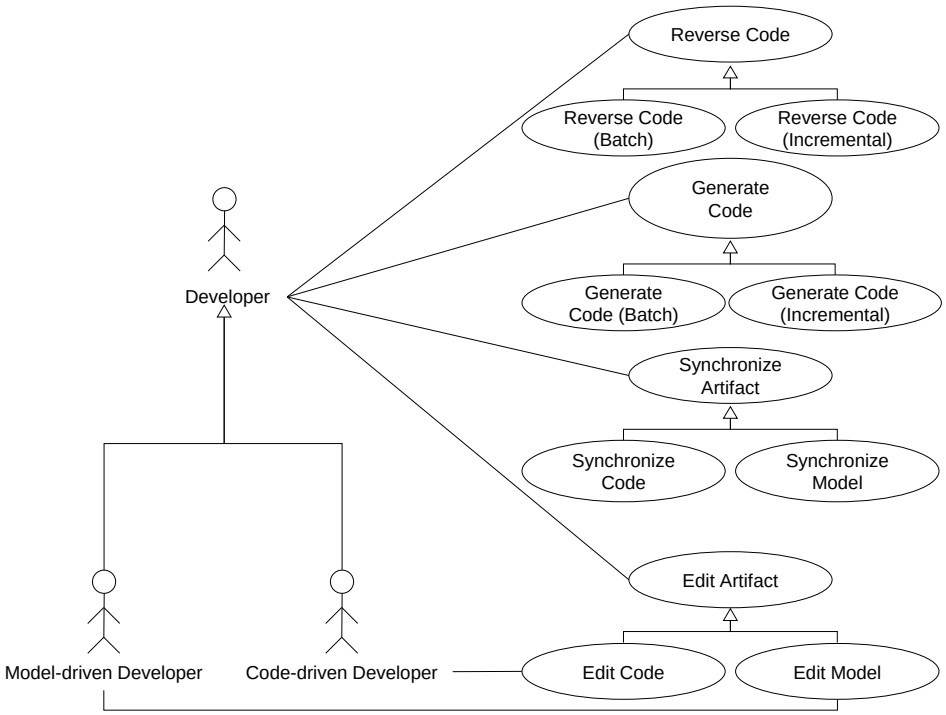


Figure 4.1: Use-cases of IDE for model/code edition and synchronization

Definition 9 (Baseline artifact) *A baseline artifact is an artifact which may be edited manually. All other artifacts are produced or updated from the baseline artifact through a process. Manual edition of artifacts other than the baseline artifact is forbidden.*

Two primary actors, called *model-driven developer* and *code-driven developer*, are introduced. The main difference between them is what they consider as the baseline artifact.

Definition 10 (Model-driven developer) *A model-driven developer is an actor in a software development process for whom the baseline artifact is the model.*

In other words, for a model-driven developer, only the model should be edited manually. The code must always be produced from the model automatically by some process that guarantees that the code is consistent with the model. A software architect is a kind of the model-driven developer who edits the model to specify the architecture of the system. An architect presumes that the reference for the architecture of the system should be specified as a model.

Definition 11 (Code-driven developer) *Code-driven developer is an actor in a software development process for whom the code is the baseline artifact.*

A programmer is a specialization of the code-driven developer. Indeed, programmers may modify the code, such as editing method bodies. The code is then the main reference for the implementation of methods.

4.1.2 Main use-cases of IDE for collaboration

In this section we propose a generic IDE with the main use-cases that represent functionalities required by our model-code synchronization approach. Figure 4.1 shows a UML use-case diagram of the IDE and associations to the actors.

There are some use-cases for manual edition of artifacts. The `Edit-Artifact` use-case implies that the IDE must have some tool to let the developer manually edit an artifact. The `Edit-Model` and `Edit-Code` use-cases are specializations of the `Edit-Artifact` use-case where the artifact is the model or code.

There are also some use-cases related to the synchronization of artifacts. The `Synchronize-Artifact` use-case is the synchronization of two artifacts by: (1) comparing them, (2) updating each artifact with editions made in the related artifact, and (3) reconciling conflicts when appropriate. The `Synchronize-Model` and `Synchronize-Code` use-cases are specializations where, respectively, the model or the code are the artifacts being synchronized.

Definitions of batch and incremental code generation The `Generate-Code` use-case is related to forward engineering. It is the production of code in a programming language from a model. In this thesis, we assume that the model is syntactically correct and the validation of model is out of the scope of this thesis. The developer can either use `Generate-Code (Batch)` or `Generate-Code (Incremental)`.

Definition 12 (Batch code generation [Giese 2006]) *Batch code generation is a process of generating code from a model, from scratch. Any existing code is overwritten by the newly generated code.*

Incremental code generation is a specialization of incremental model transformation, which is defined in [Giese 2006] as model transformation that does not generate the whole target model from scratch but only updates the target model by propagating editions/changes made in the source model. Incremental code generation is defined in this thesis as follows:

Definition 13 (Incremental code generation) *Incremental code generation is the process of taking as input an edited model, and existing code, and then updating the code by propagating model changes to the code.*

Example of batch versus incremental code generation To illustrate the differences between batch and incremental code generation, let's look at a simple example in Fig. 4.2. We assume that the model M and the code C are initially consistent. An y attribute is then added to the *Capsule* class of M and a *test* method added to the code. In the step 2a, using the batch code generation reproduces the *Capsule* class in the code C_a. This latter has the y attribute updated from the model but the previously added *test* method is lost by the batch code generation. If the incremental code generation is used in the step 2b, the resulting code C_b preserves the *test* method.

Definitions of batch and incremental reverse engineering The `Reverse-Code` use-case is related to reverse engineering. `Reverse-Code` is the production of a model from code. As mentioned in the assumption in Section 1.3, the model contains more information than the code since some information in the model is used for model-based activities such as security analysis or performance analysis. Therefore, it is impossible to reconstruct the

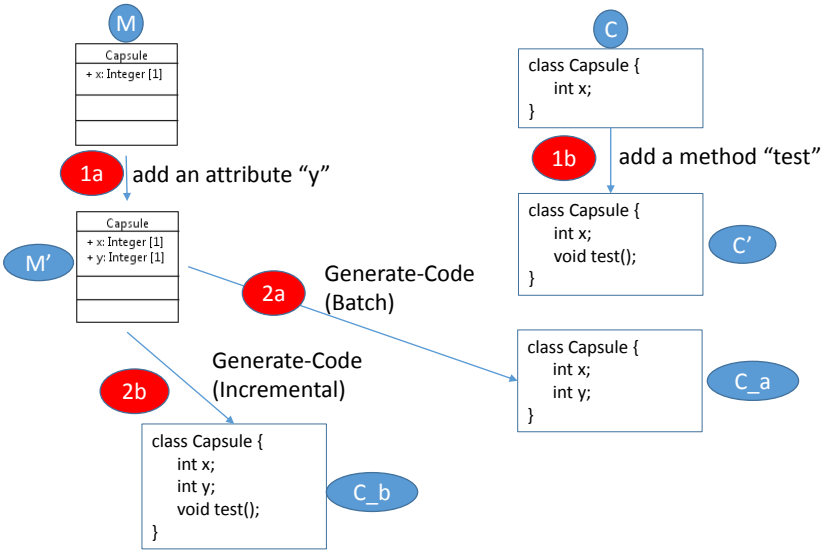


Figure 4.2: Batch and incremental code generation

entire model including information not related to code generation. In this thesis, we assume that there is no loss of information when reversing code to the *code generation-related part*. If some information not related to code generation exists in the model, the `Reverse-Code (Incremental)` use-case should be used for preserving such information during reversing. The developer can either use `Reverse-Code (Batch)` or `Reverse-Code (Incremental)`, which are defined in this thesis as follows:

Definition 14 (Batch reverse engineering) *Batch reverse engineering is a process of producing a model from code, from scratch. The existing model is overwritten by the newly produced model.*

Definition 15 (Incremental reverse engineering) *Incremental reverse engineering is the process of taking as input a edited code, and an existing model, and then updating the model by propagating editions in the code to the model.*

For readability, in this thesis we will sometimes designate batch and incremental as modes of code generation/reverse; e.g. we say that we generate code in batch mode from a model.

The use-cases are generic. They do not depend on any particular approach or tool. Therefore, software developers can choose the approach or tool that suits better his/her development preferences best.

In the next section, the use-cases of the IDE are integrated into multiple processes that cover model-code synchronization in several scenarios. The latter correspond to behaviors performed by both kinds of actors, i.e. model-driven developers and code-driven developers.

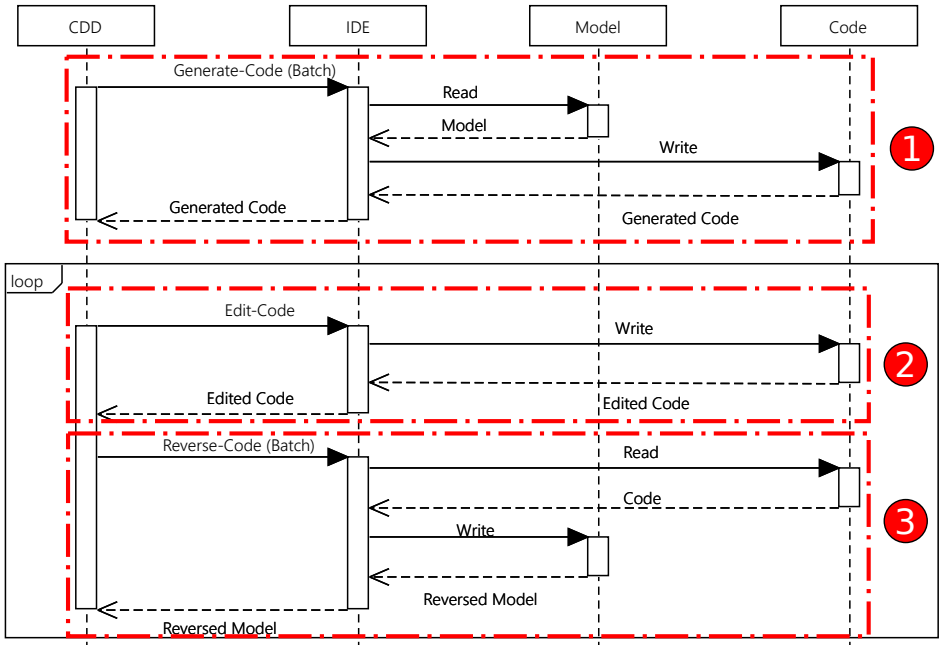


Figure 4.3: Synchronization process for scenario 1, in which only code is edited (CDD = Code-Driven Developer). The API calls for Model and Code are represented generically as "Read" and "Write".

4.2 Processes to synchronize model and code

This section describes multiple processes for synchronizing model and code in different scenarios. The scenarios are differentiated according to the type of developers, i.e. the type of actors, working on the system to be developed.

For each of the scenarios, we assume that the common starting point of each process is a complete model. Therefore, if legacy code exists outside of the model, it must first be reversed back before the processes described below can be applied. This can be done with the `Reverse-Code (Batch)` use case of the IDE proposed in Section 4.1.2.

Each of the following subsections describes a scenario and the process associated with it for model-code synchronization. The last subsection discusses how the proposed processes can be applied to reconcile conflicts between editions made the model and the code.

4.2.1 Scenario 1: code-only editions

The assumption in the scenario 1 is that the code is the only baseline artifact. In other words, only the code may be edited manually. We assume that during development, the model needs to be synchronized sporadically with the code. This scenario is usually more appropriate for the code-driven developer.

Figure 4.3 shows a sequence diagram that illustrates the process for synchronizing code and model in the scenario 1. The general approach is to always overwrite the model with a new model produced from the edited code.

In Figure 4.3, the code-driven developer interacts with the IDE which then writes and reads from code and model. Keep in mind the general assumption that the starting point

of the process is a model. Note that messages between the code-driven developer and the IDE are the use-cases of the IDE proposed in Section 4.1. In Figure 4.3, there are some markups to highlight the general steps of the process. The general steps of the process are described as follows:

- Scenario 1 synchronization process general steps -

Step 1 To begin implementation, the code is generated in batch mode from the model. The developers then work with this generated code.

Step 2 The code is edited.

Step 3 After the code has been edited, it is reversed in batch mode, i.e. the existing model is overwritten by a new model reversed from the code.

Obviously code can evolve several times through successive editions. After each change cycle, the code can be reversed. Therefore steps **(2)** and **(3)** may be repeated multiple times in this process. Since code is the baseline artifact here, we only need to overwrite the model by batch reverse for both artifacts to be synchronized. Incremental reverse engineering can also be used in place of the batch reverse. For a model that only contains information related to code generation, using incremental code reverse has the same effect as batch code reverse. However, if there is a model part not used for code generation, the batch reverse might not preserve such part. Therefore, the incremental reverse engineering should be used in this case.

4.2.2 Scenario 2: model-only editions

Scenario 2 is the opposite of the scenario 1: the assumption here is that the model is the only baseline artifact. That is, only the model may be edited manually. This scenario is appropriate for the model-driven developer.

The process for scenario 2 is similar to the process for the scenario 1. Therefore, the sequence diagram for this scenario is put in Appendix B on page 131. In this scenario, code is first generated in batch mode from the model. The model is edited. After each edition of the model, code is generated in batch mode, i.e. the existing code is overwritten by new code generated from the model.

4.2.3 Scenario 3: concurrent editions

Note that the processes proposed for the scenario 1 and scenario 2 are classic cases of forward and reverse engineering, rather than fully-fledged round-trip engineering with a need for synchronization. Indeed, code is produced from model in batch mode. When it is reversed, it is revered to the model in batch mode. There is no need for additional synchronization strategies, since only either the code or the model is the baseline artifact, i.e. only one is edited manually and there are no concurrent editions being made to the two artifacts. Therefore batch code generation and reverse are sufficient for synchronization.

However, in the scenario 3, there is no unique baseline artifact; both the model and the code may be edited manually. Therefore, they may evolve concurrently during development activities and synchronization issues will occur. This scenario tackles the problem where model-driven and code-driven developers are working together on the same system.

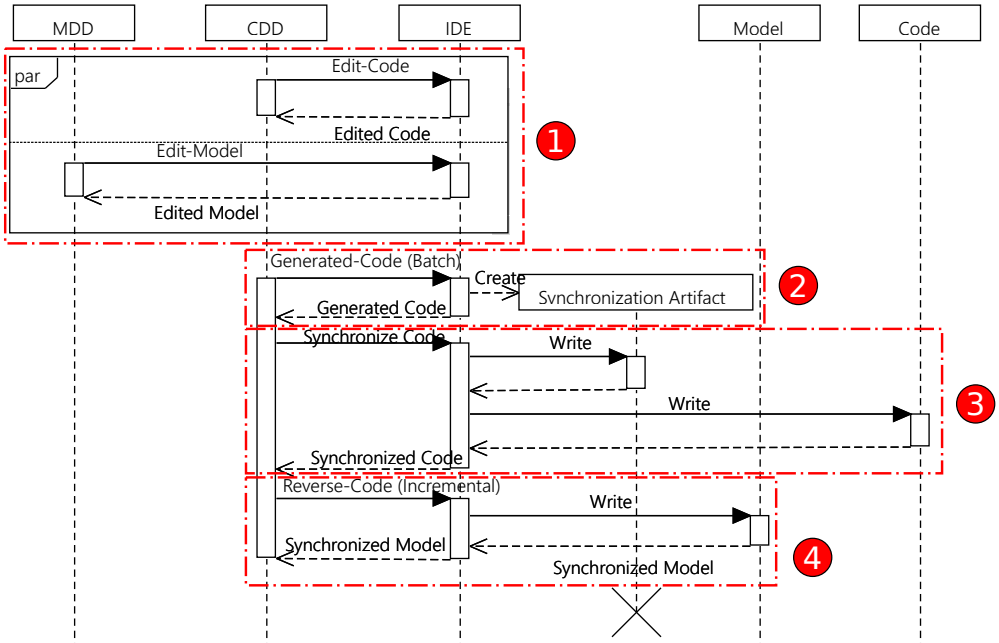


Figure 4.4: Synchronization process for scenario 3 using the first strategy, in which the model and the code are concurrently edited with code as the synchronization artifact (CDD = Code-Driven Developer, MDD = Model-Driven Developer). The API calls for Model and Code are represented generically as "Read" and "Write".

We propose two synchronization strategies for this scenario. The general approach behind our synchronization strategies is to represent one artifact in the language of its corresponding other artifact. These two can then be compared. For this, we define a concept of a *synchronization artifact*:

Definition 16 (Synchronization artifact) *An artifact used to synchronize a model and its corresponding code is called a synchronization artifact (SA). It is an image of an artifact, either model or code, in a different representation. In this context, an image I of an artifact A is a copy of A obtained by transforming A to I. A and I are semantically equivalent but are specified in different languages.*

For example, an SA can be code that was generated from the edited model in batch mode. In that case, it is code that represents an image of the edited model.

Using the concept of SA, two strategies are proposed in this thesis: one in which the SA is code, and the other in which the SA is a model. We propose two strategies so the developer can choose to either use the `Synchronize-Code` or `Synchronize-Model` use-cases of the IDE. The choice may be determined by preferred development practices or the availability of suitable tools (e.g. the programmer may prefer to synchronize two artifacts, both represented in the same programming language, because he prefers to work exclusively with code). Figure 4.4 shows the first synchronization strategy based on using code as the SA. The highlighted general steps of the process are described as follows:

- Scenario 3 synchronization process steps -

Step 1 Both the model and code may be edited concurrently. (To simplify Figure 4.4, we

don't show the Read and Write interactions for this step.) After both artifacts have been edited concurrently, we need to synchronize them.

Step 2 First we create a synchronization artifact from the edited model by generating code in batch mode. This synchronization artifact is code and it is an image of the edited model.

Step 3 The synchronization artifact is synchronized with the edited code. Since the synchronization artifact is code itself, this step is done with the `Synchronize-Code` use-case of the IDE.

Step 4 Once synchronization artifact and edited code are synchronized, the former is reversed incrementally to update the edited model.

The second strategy, based on using model as the synchronization artifact, is the opposite of the first strategy. In the second strategy, the synchronization artifact is obtained by reversing the edited code in batch mode. Afterwards the synchronization artifact is synchronized with the edited model. Finally, we generate code incrementally from the synchronization artifact to update the edited code.

We propose two strategies based on the preferences of the developers. They may even use both strategies, successively, as a kind of hybrid strategy. This may be useful when developers want to synchronize parts of the system using one strategy, and other parts using the other strategy. For example, they may choose to synchronize method bodies using strategy 1, where the synchronization artifact is code. Then strategy 2, in which the synchronization artifact is a model, is used to synchronize architectural elements of the system.

Synchronization example with the first strategy Fig. 4.5 shows how the first strategy works to synchronize the *Capsule* class of the *Edited model* with that of the *Edited code*. The step 2 in the strategy 1 in Fig. 4.4 produces the *Synchronization artifact*, which is code in this case, by using the batch code generation. This synchronization artifact contains updates from the model including the *y* added attribute. The synchronization artifact and the edited code, which contains the *test* added method, are then merged with each other by using the *Synchronize-Code* use case defined previously. The result is that *Synchronized code* contains both editions, the added method and attribute, made in the edited model and edited code previously, The last step updates the added method *test* in the synchronized code back to the model by using the incremental reverse engineering. At the end of the synchronization, the modifications of the *Capsule* class at the code and the model level are reconciled and model and code become consistent.

Synchronization example with the second strategy The edited model and code in the *Capsule* example in Fig. 4.5 can also be synchronized by using the second strategy as in Fig. 4.6. The *Synchronization artifact* created in this case by using the batch reverse engineering is a model. The *Synchronize-Model* use case in the step 3 then merges the edited model and the synchronization artifact into the *Synchronized model* that contains the added method *test* and attribute *y*. Note that, the synchronization model use case only reconciles the code generation-related part of the edited model with the synchronization artifact. The synchronization between models as described in Subsection 1.3 is in charge of propagating model editions from the code generation-related part of the model made by the `Synchronize-Model` use case to the other information part.

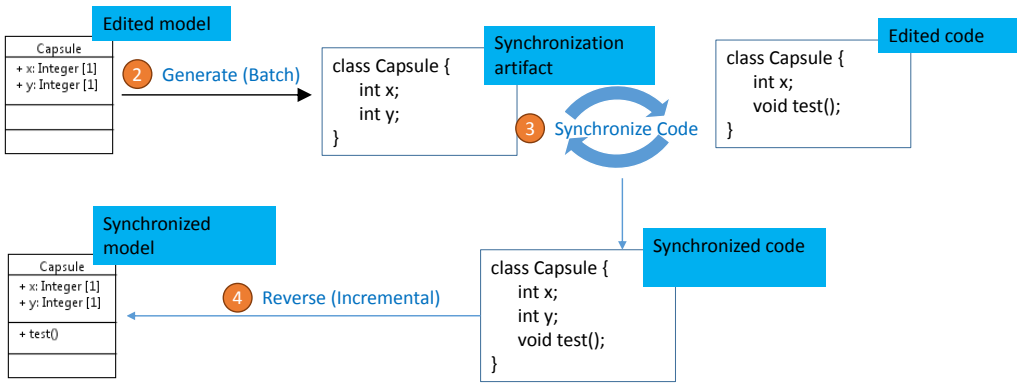


Figure 4.5: Example of synchronizing concurrently modified model and code using the first strategy

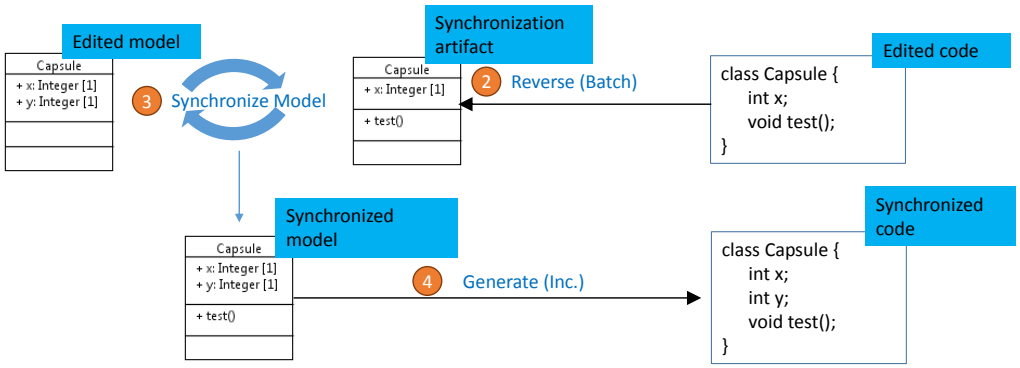


Figure 4.6: Example of synchronizing concurrently modified model and code using the second strategy

4.2.4 Discussion

This subsection discusses the following questions related to the proposed processes:

- How are conflicts tackled in concurrent editions?
- Do the processes allow multiple concurrent editions made by several model-driven developers or code-driven developers on the same artifact?

Conflicts between editions made in model and code occur when propagation of editions from the model made by a model-driven developer to the code is in conflict with code editions made by a code-driven developer in the code. An example of a conflict is that the same class is renamed in the model and deleted in code. Since the purpose of the processes is to provide a synchronization mechanism while still allowing the developers to work with their preferred choice of tools and development artifacts, we also provide such capability when dealing with conflicts. The reconciliation can be done following one of the following ways:

- At the code level: A code-driven developer can reconcile conflicts while working at the code level and then automatically propagate modifications made in the code during the reconciliation of the conflicts to the model. Using the first strategy of the synchronization processes, the reconciliation of the problem of concurrent modification of model and code with conflicts becomes the reconciliation of modifications made to the same code, *Generated Code* and *Edited Code* in the step 3 in Fig. 4.4. An example of reconciliation is described below. Such code-code reconciliation is supported by the *Synchronize-Code* use case defined in Section 4.1.
- At the model level: A model-driven developer can reconcile the conflicts while working at the model level and then automatically propagate modifications made in the model during the reconciliation of the conflicts to the code. Such way is similar to the other one at the code level. The problem of dealing with conflicts here becomes the model-model reconciliation supported by the *Synchronize-Model* use case defined in Section 4.1

Let's examine a realistic example of conflicts. The model and the code of a system initially contain three classes, namely *ClassA*, *ClassB*, and *ClassC*. The model and the code are stored in a remote Git repository. A model-driven developer (MDD) and a code-driven developer (CDD) collaborate with each other for developing the system. Each developer has a copy of the model and the code and can push as well as get data from the remote repository. Suppose that the MDD renames *ClassB* to *ClassD*, regenerates the code using the batch or incremental code generation, and pushes the files containing the renaming-affected editions with a change log. Note that, the push and the change log are typically used in version control systems such as Git. At this point, the pushed code is the *Synchronized Artifact* in Fig. 4.2.3 since it is generated from the model edited by the MDD.

In the meantime, the CDD deletes *ClassB* at the code level. The code modified by the CDD now becomes the *Edited Code*. The CDD now gets the pushed code from the remote repository and merges the locally edited code and the remotely edited code (`Synchronization Artifact`) as the step 3. The CDD reads the change log and sees a conflict that the class that he/she has just deleted is renamed. This conflict is also detected by Git. He/she then decides to either reject the edition of the MDD (so to delete the class)

or accept it (so to keep the class). Whatever decision is taken on, the CDD only works at the code level and does the step 4 to automatically propagate the edition decision to the model. Eventually, the CDD pushes the synchronized model and the code to the remote repository. The conflicts are reconciled without requiring the CDD to change her/his preferred practice of working with code.

Multiple editions by multiple developers In software development of a project, there might be multiple model-driven developers editing the same model element and multiple code-driven developers editing the same code element. In this case, the synchronization must deal with concurrent editions made in the model and the code by multiple developers. To deal with this issue, each developer of a certain type (model-driven or code-driven) should repeat the strategy appropriately adapted to his preferred practice. A model-driven developer uses the second strategy whenever he/she wants to synchronize his/her editions in the model with the editions made in the model by other model-driven developers and in the code by other code-driven developers. Similar procedure is also applied to code-driven developers, who use the first strategy to synchronize with other editions in the model and the code by other developers.

In the next section we present an approach that reuses different existing technologies and tools to implement the proposed IDE and synchronization processes.

4.3 An Implementation: synchronizing UML models and C++ code

This section describes the instantiation of the proposed model-code synchronization approach for the particular case of OMG standard UML 2.5 and C code. We focus on elements of the UML meta-model that are used to model a low-level software design, represented in a class diagram. UML elements that are considered are classes, data types, primitive types, properties, operations, opaque behaviors, value specifications, template signatures, template parameters, and template bindings. At the C++ side, we consider a number of elements that are present in the runtime system exposed in Section 4.4.2. Some C++ elements that are considered are classes and structs, type definitions, C primitives, attributes, methods and method bodies, modifiers for attribute and method, includes, macros, and forward declarations, pointers and references, function pointers, variable initialization, class and function templates.

The lack of tools could be a risk to the usability and validity of the approach. It is critical that tools are readily available or implementable with moderate effort. We examine an Eclipse-based implementation of the IDE proposed in Section 4.1.2. The implementation is used in the synchronization processes proposed in Section 4.2. The use-cases of the IDE are implemented with some Eclipse technologies. More information on the technologies is available on the Eclipse Projects website [Eclipse Foundation].

Eclipse CDT is an IDE for C++ development. It is used in the `Edit-Code` use-case. Eclipse **Papyrus** is used for the `Edit-Model` use-case. Papyrus is an open-source UML modeler that uses the Eclipse Modeling Framework (EMF) implementation of the OMG-standard UML 2. The EMF implementation of UML 2 stores UML models as XMI files. Papyrus supports UML profiles for domain-specific modeling. We use the UML profile of Papyrus dedicated to C++. The profile has a number of stereotypes to model C++ elements that cannot be created directly in UML (e.g. function pointer).

Formerly, Papyrus supports the `Generate-Code (Batch)` from UML to C++, We developed the `Generate-Code (Incremental)` and `Reverse-Code` use-cases for Papyrus. The batch modes of these use-cases do not need additional technologies to implement. For the use-cases `Generate-Code (Incremental)` and `Reverse-Code (Incremental)`, we choose to listen to modification events in the model and code respectively. Listening to modification events is one possible approach in incremental model transformation [Kusel 2013]. The **Viatra** API is used to listen to such events in the model. The Eclipse CDT API is used to listen to modification events in the code. These lists of events are used to either generate code or reverse code incrementally.

EMF Compare is used for the `Synchronize-Model` use-case when models are implemented with EMF. Eclipse CDT is used for the `Synchronize-Code` use-case with its built-in C++ features.

By implementing the proposed model-code synchronization approach with several Eclipse technologies, we faced some technical issues. The following paragraphs describe the difficulties we faced, and how limitations of Eclipse technologies were leveraged.

Eclipse CDT Modification Events Editions made to the code trigger three kinds of Eclipse CDT modification events: `ADDED`, `REMOVED`, and `CHANGED`. Events `ADDED` and `REMOVED` mean addition and deletion of classes, attributes or methods in the code. Event `CHANGED` is the update of elements in code including (1) renaming attributes, classes, or methods, (2) changing the type of attributes, parameters, or (3) changing the behavior of methods. For attributes and methods, CDT does not always trigger an appropriate `CHANGED` event, rather the framework triggers an `ADDED` event, followed immediately by a `REMOVED` event. Many events had to be filtered and combined as appropriate `CHANGED` events, according to heuristics.

EMF Compare Stereotype Comparison EMF Compare is used to compare UML elements that may be stereotyped. The stereotypes are those of the UML profile for C++ elements modeling, within Papyrus. We adapted EMF Compare for stereotype application comparison. Indeed, EMF Compare always detects differences between stereotype applications if they do not have the same XMI identifier, even if semantically they represent the same information. The EMF Compare API offers post-comparison hooks to filter differences according to the user's needs. We thus filtered differences between stereotype applications, when they are semantically equal.

Eclipse CDT Code Comparison The Eclipse CDT code compare feature only computes differences between two codes. It does not consider that the two codes may share some history. It is therefore difficult to know what to compare for elements that underwent editions such as renaming, either in the edited UML model (reflected in the synchronization artifact) or the edited C++ code. User knowledge is necessary to solve this issue with the current implementation. We help the user by providing a list of filtered editions in both the UML model and the C++ code.

In conclusion the implementation of the approach, with Eclipse technologies, was not straightforward. For synchronization strategy 1, using code as synchronization artifact, heuristics and user intervention had to be introduced. The limitations of Eclipse CDT prevented the full automation of this strategy. For synchronization strategy 2, using model as synchronization artifact, tools like EMF Compare had to be extended in order to fully

automate this strategy.

The next section describes our experimentation performed using the above implementation of our synchronization solution to evaluate the proposed methodological pattern.

4.4 Experiments and Evaluations

As a reminder, the purpose of the proposed model-code methodological pattern is to address the challenge RC1 and satisfy the requirements for model-code synchronization as shown in Table 2.1 on page 14. First, in the forward direction, the R1 (model modification propagation) and R4 (code modification preservation) requirements are satisfied if the batch and incremental code generation are correct, respectively. Secondly, in the reverse direction, the R2 (code modification propagation) and R3 (model modification preservation) requirements are met if the batch and incremental reverse engineering are correct, respectively. Lastly, the R5 (concurrent modification) requirement is validated if the proposed processes for model-code synchronization functions correctly.

This section reports our experiments with the proposed model-code synchronization approach and its implementation based on Eclipse technologies to assess the requirements for the case of synchronization of UML and C++. Two experiments have been conducted in order to assess the proposed methodology and its applicability to the development of a realistic system. In the following subsections, we first describe the results of multiple simulations designed to test the correctness of the batch code generation and reverse engineering (see 4.4.1.2), of the incremental reverse engineering (see 4.4.1.3), and of the incremental code generation and synchronization processes for concurrent modifications (see 4.4.1.4). Next, we report on a realistic case-study intended to help us assess certain scalability and usability aspects (see 4.4.2).

4.4.1 Simulations to assess synchronization processes

Simulations have been conducted to test that the implementation of our proposition respects the round-trip engineering laws [Foster 2007] *right-invertibility* and *left-invertibility*. These laws are stated as follows:

Law 1: Right-invertibility means that not editing the code (or model, respectively) shall be reflected as not editing the model (respectively code). If no editions are made to the code, the model used for generating the code and the model received by immediately reversing the generated code must be the same.

This law is strong since there can be more than one correct way to reverse a code element. For example, a pointer in a class A to a class B in code can be reversed to model either as an attribute of the class A or an association from A to B. Furthermore, an interface in C++, which is a class whose methods are *pure virtual* can be reflected in UML as either a UML class or a UML interface. For those cases, the developments of code generation and reverse engineering are required to be tightly related to each other. It is a limitation of the evaluation imposing on code generation and reverse engineering. However, the tight relationship between them helps developers avoid ambiguities between the model elements created by model-driven developers and the elements created by reversing code elements written by code-driven developers. To deal with the ambiguous cases as above, a stepwise comparison of two models in determining whether they are the same or not is based on comparing both syntactics and semantics of the two models. Table 4.1 shows pairs

Table 4.1: Model elements with same semantics

Model element 1	Model element 2
Interface	Class with all abstract operations
Aggregation association from A to B	Shared attribute in A typed by B
Composition association from A to B	Composite attribute in A typed by B
Attribute tagged by a pointer stereotype	Shared attribute
Attribute tagged by an array stereotype	Composite attribute with multiplicity
Attribute tagged by a pointer and an array stereotype	Shared attribute with multiplicity

of model elements that have the same semantics. According to the table, an interface of the model for code generation is equivalent to a class with all abstract operations of the model created by reverse engineering.

Law 2: Left-invertibility means that all editions on the code are captured and correctly propagated to the model so that the edited code can be fully recreated by applying code generation to the updated model.

Note that this law is correct if the batch code generation, the batch and incremental reverse engineering are correct.

In the following sections, the model generator used for the simulations is presented first. Then, the simulations performed for both above laws are described. Finally, we discuss the simulation of the process defined for scenario 3 (Section 4.2.3), in which both the model and the code are edited concurrently.

4.4.1.1 Randomly generated UML models

In the simulations, random UML models are generated with a configurable model generator we implemented. The generated models contain C++ features represented via UML. The generator can be configured to generate a desired average number of each type of C++ feature to be represented as a UML element.

Three packages are generated for each model. Each package contains 60 classes, and, on the average 10 enumerations, 10 structures, and 10 function pointers. Class members include methods with parameters, and attributes. Types of parameters and attributes are chosen randomly. Methods have randomly generated bodies that use other classes. Attributes have randomly generated default values of the appropriate types.

Relationships between classes are also generated: associations, inheritances, and dependencies. When dependencies are generated, the generator enforces that the source class of the dependency has a method that uses the target class of the dependency.

4.4.1.2 Simulation for Law 1 right-invertibility

In the first simulation, Law 1 is evaluated. The procedure for the simulation is shown in Figure 4.7.

The general idea behind the simulation procedure is to do a full round-trip of the UML model randomly generated in step **(1)**. The round-trip of the model is done through steps **(2)** to **(3)**. The randomly generated model is compared with the reversed model after the round-trip. This is done in step **(5)**.

A full round-trip is also done for the C++ code that is generated from the original

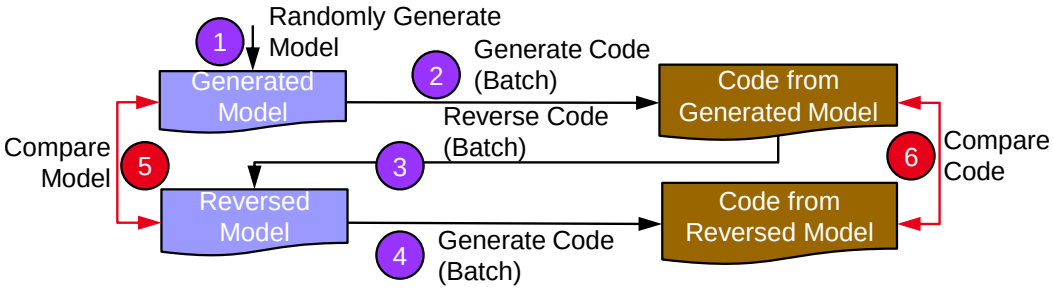


Figure 4.7: Simulation 1 for Law 1 (right-invertibility)

randomly generated model in step **(2)**. The round-trip is done through steps **(3)** to **(4)**. The code generated from the original model is compared with code generated from the reversed model in step **(6)**.

Code generation and reverse are done in batch mode since no editions are made to the models or code in this simulation. Comparison of models is done by first syntactically comparing the number of each type of elements and then semantically using Table 4.1 and EMF Compare. Comparison of code is done by using the built-in code comparison tool in the Eclipse CDT.

200 models, with random numbers of elements were created by the generator. We limited ourselves to 200 models for practical reasons. These same models were later used for a simulation that involved some manual editions (presented in the next subsection).

After launching the simulation for the 200 models, no differences were found during model comparison and code comparison. This result assesses that the tooling of our model-code synchronization approach with respect to Law 1 of round-trip engineering. The limitation of the result is that the generated models share some properties since they are produced by the same generator. To increase the confidence, the approach is applied to a case study that is presented in the subsection 4.4.2.

4.4.1.3 Simulation for Law 2 left-invertibility

In the second simulation, Law 2 was evaluated. The procedure for the simulation is shown in Figure 4.8. The general idea behind the simulation procedure is to do a full round-trip of a randomly generated UML model, but with editions introduced to the generated C++ code. As shown in Figure 4.8, the procedure to test our tooling consisted of the following steps:

Step 1 A UML model is randomly generated and becomes the **generated model**.

Step 2 Through batch code generation, we obtain **code from generated model**.

Step 3 The code produced from the generated model is then edited. We thus obtain **edited code**. The different kinds of editions will be described when we discuss the results of this simulation.

Step 4 The **edited code** is reversed incrementally to the **generated model**. The **generated model** then becomes an **updated model**.

Step 5 The **edited code** is also reversed in batch mode to a **model from edited code**. This model is an image of the edited code.

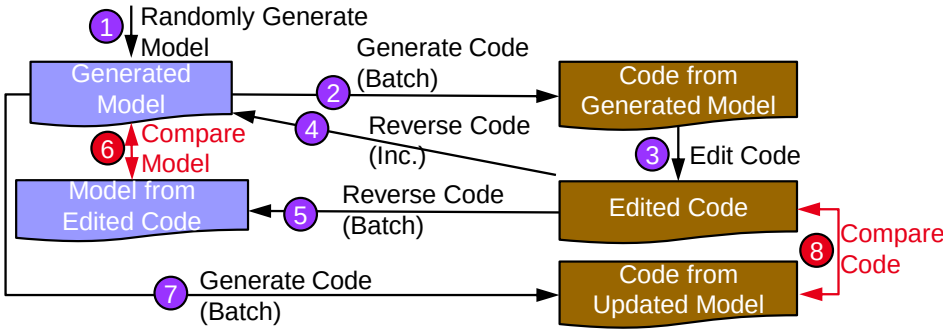


Figure 4.8: Simulation 2 for Law 2 (left-invertibility)

Table 4.2: Editions and the modifications events they trigger (A = `ADDED`, C = `CHANGED`, R = `REMOVED`)

Edition kind	A	C	R
Renaming attributes of all classes	0	1903	0
Renaming methods of all classes	0	1197	0
Deleting attributes	0	0	46
Adding attributes	25	0	0
Adding methods	10	0	0
Changing method body	0	30	0
Manual editions	39	34	30

Step 6 The **updated model** (the previously **generated model**) is compared to the **model from edited code** (image of the edited code).

Step 7 Afterwards, we also generate in batch mode **code from the updated model**.

Step 8 The **code from the updated model** is compared to the **edited code**.

The simulation is run for 200 randomly generated models. We limited ourselves to 200 models for practical reasons because in step **(3)**, some of the editions have to be done manually to emulate real development conditions. Therefore, the limit of 200 models is purely based on practical concerns.

During the simulations, the code generated from each model undergoes seven kinds of editions independently. The kinds of editions performed are listed in Table 4.2. As its name suggests, only "Manual editions" were done manually by a developer, to emulate actual development conditions.

As a reminder, the editions trigger three kinds of Eclipse CDT modification events: `ADDED`, `REMOVED`, and `CHANGED`. Table 4.2 shows the average number of `ADDED`, `REMOVED`, and `CHANGED` events that were triggered by each of the seven kinds of editions. For example, there are 1903 `CHANGED` events related to attribute renaming. Two kinds of editions are not listed in the table because they trigger the same number of events as other editions. Indeed, changing modifiers of attributes/methods is equivalent to renaming attributes/methods. Deleting methods is equivalent to deleting attributes.

For all 200 models, no differences were found during model comparison and code comparison. This result assesses that the tooling of our model-code synchronization approach

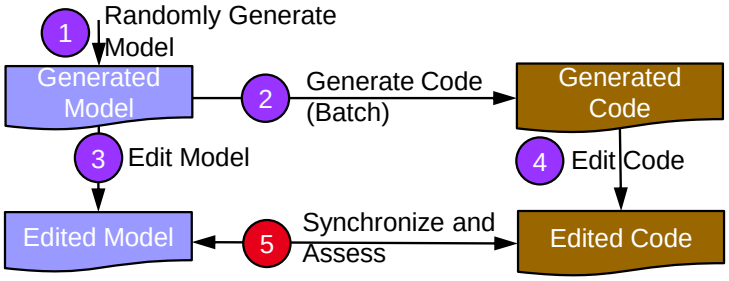


Figure 4.9: Simulation 3 for concurrent editions case

with respect to Law 2 of round-trip engineering.

4.4.1.4 Simulation for concurrent edition of model and code

A third simulation aims at emulating the process described in Figure 4.4 of Section 4.2.3, in which the model and the code are concurrently edited. A simplified representation of the procedure of simulation 3 is shown in Figure 4.9. The simulation is executed for 200 randomly generated UML models and their generated C++ code.

The idea behind the procedure in Figure 4.9 is to simulate editions in the original randomly generated model, and its corresponding generated code (step **(1)** to **(4)**). To simplify the simulation of the artifact synchronization use-case, the simulator only introduces attribute renaming on the model. On the code, the simulator only makes editions to method bodies, which is a limitation of the experimentation based on the simulations. More experiment about modifications with conflicts and reconciliation is scheduled for future work. Edited model and code must then be synchronized and then we must assess that they are indeed synchronized. This is done in step **(5)**.

To synchronize edited model and edited code, first we use the strategy based on code as the synchronization artifact. The simulator propagates all method body editions from the edited code to the synchronization artifact. The simulator then propagates all attribute renaming from the synchronization artifact to the edited code. Next, the synchronization artifact is reversed incrementally to produce the edited model. At this point the model and the code are considered synchronized.

To assess that both synchronized model and code are images of one another, the synchronized code was reversed in batch mode to a new model. The new model was then compared with the synchronized model. No differences were found during these simulations. We also generated new code in batch mode from the synchronized model. The new code was compared to the synchronized code. No differences were found during the comparison.

The same simulation is repeated for the case where the model is used as synchronization artifact. Again the synchronized code and model were images of one another.

Once the tooling was verified through the three simulations, we applied it to the development of a real-world system. The results of this experiment are discussed in the next section.

4.4.2 Papyrus-RT runtime case-study

In Section 4.4.1 the approach was tested for UML and C++ synchronization, using simulations. In this section we describe the application of the full synchronization approach to

Table 4.3: Differences in Papyrus-RT runtime versions

Original	Object-oriented
Directives to control compilation of OS-dependent code	Abstract OS-independent classes inherited by OS-dependent classes with implementation
Variables, functions and type definitions outside of class	Refactored as static entities inside a class. Visibility defined according to scope of original entities.

a case-study. The intent here was to evaluate:

- The usability of the proposed synchronization approach
- The scalability of the tooling to a real system

The case-study is related to the development of the runtime underlying Papyrus-RT [CEA 2016]. This latter is an open-source custom modeling tool, based on Papyrus, that supports the UML extensions for the design of event-driven real-time systems (UML-RT). Papyrus-RT features full automated code generation for UML models. The generated code executes within a corresponding runtime environment, which provides C++ realizations of the high-level concepts defined in UML-RT.

At the beginning the resource available to develop the aforementioned runtime was an experienced C++ programmer. For the pragmatic reasons of project management such as time restrictions, it was decided to implement the runtime in C++ without the benefits of models. However, later it was concluded that even if the result was satisfactory, it would be beneficial to exploit the advantages that MBSE provides. Thus, an MBSE approach would improve both the maintainability and the evolvability of the runtime. Moreover, as development progressed, the project was expanded to include new developers, who are proponents of MBSE and are eager to work with models.

As a result, this project represented an ideal case study to assess the practicality of the proposed approach. In particular, we were interested in determining its usability and scalability

The runtime was originally implemented as an open-source plain C++11 project. Most of its architecture is object-oriented. The runtime has 65 classes and 14,945 lines of code. Other than containing typical entities found in object-oriented architectures, the runtime uses C/C++ features such as type definitions, templates, pointers, references, function pointers, and variadic functions to name a few. These features are supported by the reverse and code generation tools in Papyrus, coupled with the C++ UML profile.

In order to use our model-code synchronization approach, the original Papyrus-RT runtime had first to be reversed to a UML model. This step was crucial because the original runtime contained some elements that were not object-oriented. Consequently, some modifications and refactorings were made to the original runtime to make it fully object-oriented. This enabled it to be modeled entirely in a language like UML. Table 4.3 shows the two main differences between the original runtime, and the revised object-oriented runtime.

The reverse in batch mode of the object-oriented runtime takes about 12 seconds. All 65 classes were reversed, with all of their attributes and methods. Code generation in batch mode of the entire reversed UML model takes about 5 seconds and produces 22,053 lines

of code. The difference in the number of lines of code is due to automatically generated documentation comments. The generated runtime compiles and the updated existing unit tests pass when applied on the runtime compiled from generated code.

Once the runtime has been reversed, our model-code synchronization processes could be used. We noticed that using the proposed approach introduced MBSE style development for the Papyrus-RT runtime. This brought about several advantages as several manual tasks were automated:

- Automatic handling of relationships between model elements, i.e. association, dependency, inheritance
- Automatic generation of includes and forward declarations in the code that avoids cyclic dependencies
- Graphical representation of architecture in automatically updated UML diagrams (feature of Papyrus)

In applying the proposed approach to achieve a collaborative development of the runtime system, we found that the only real difficulty faced was initializing the synchronization processes. This required the reverse engineering of the original runtime, with some non-object-oriented code, into an object-oriented UML model. The reverse was successful following some modifications and refactorings. Once the full model-code synchronization process was in place, we were able to use it and develop concurrently both the UML model and its generated C++ code.

4.5 Summary

This chapter presents a generic model-code synchronization methodological pattern. It can be used to solve the issue of model-code synchronization when artifacts co-evolve. The proposed approach is designed to support a continuous collaboration between software architects and programmers allowing each to use the language and tools of choice. An Eclipse-based implementation of the approach was presented for synchronizing UML models and C++ code.

Multiple simulations were conducted to validate our synchronization approach with respect to both laws of round-trip engineering and the artifact synchronization requirements in Table 2.1. In addition, a simulation of scenarios in which both the model and the code were edited concurrently was performed, further demonstrating the viability of the approach. The approach was then applied to a real-world application: the Papyrus-RT runtime system. The experimentation showed that the main difficulty of using our approach applied to such a system was to refactor and reverse the code into a corresponding UML model without loss of information, which respect to the information containment assumption. Once the processes of our synchronization solution were bootstrapped in this way, we were able to reap the important benefits of MBSE; development was facilitated through increased automation.

Since the proposed methodological pattern is generic, it is then applicable to other synchronization scenarios with the support of the use-cases defined in Section 4.1. In the next chapter, we will show how to apply the proposed model-code synchronization methodological pattern to the problem of synchronizing UML-CS and UML-SM elements with the extended code.

Chapter 5

A bidirectional mapping and synchronization of model and code for reactive systems development

Contents

Abstract: The previous chapter presents a generic model-code synchronization methodological pattern (synchronization mechanism). This chapter shows how to use the latter to contribute to the overall approach presented in Chapter 3 for synchronizing changes made in architecture design model, specified using UML-CS and UML-SM elements, and code.

To do it, we first establish a mapping mechanism. The mapping mechanism consists of a bidirectional mapping between the architecture design model and the code, which have

a significant abstraction gap, and an in-place text-to-text transformation that acts as a preprocessing step. The bidirectional mapping includes equivalences of model elements and code elements so that the synchronization can be eased. The realization of the IDE use-cases of the synchronization mechanism such as incremental reverse engineering and code generation can be greatly simplified by using the mapping as a means. The in-place text-to-text transformation is based on a set of code generation patterns for producing code from UML-CS and UML-SM elements. We then show how the mapping and the synchronization methodological pattern are combined with each other to produce the desired requirements.

The remaining structure of this chapter is as follows: Section 5.1 establishes the bidirectional mapping. Section 5.2, 5.3 and 5.4 describe the set of code generation patterns used by the text-to-text transformation. The application of the methodological pattern using the mapping mechanism to synchronize the architecture design model and code is shown in Section 5.5. Section 5.6 presents the evaluations of the contribution related to the previously described requirements. The chapter is finally concluded in Section 5.7.

5.1 Bidirectional Mapping between architecture structure and behavior with code

In this section, we describe the bidirectional mapping for the UML-CS and UML-SM elements, and code. We present the bidirectional mapping through a producer-consumer example, whose architecture is shown in Fig. 5.1 (a), (b), and (c). This example is fictitiously created for illustration purposes only and it might not fit to realistic usages. The *p* producer sends data items to a first-in first-out component *FIFO* storing data. The *FIFO* queue has a limited size, the number of currently stored items (*numberOfItems*) and the *isQueueFull* operation for checking its fullness. The *pPush* port of the producer with *IPush* as required interface is connected to the *pPush* port of *FIFO* that provides the *IPush* interface. The producer and *FIFO* can interact with each other through their respective port. *FIFO* also provides the *IPull* interface for the consumer to get data items. *FIFO* implements the two interfaces as in Fig. 5.1 and Fig. 5.2 at lines 28-29.

The behavior of *FIFO* is described by a UML-SM as shown in Fig. 5.1 (c). Initially, the *Idle* state is active. The state machine then waits for an item to arrive at the *fifo* part (through the *pPush* port). The item is then checked for its validity before either adding it to the queue or discarding it (if the queue is full).

Table 5.1 and 5.2 show the UML meta-classes and our ad-hoc equivalent constructs in the extended language. The constructs are categorized into *structural* (four upper rows in Table 5.1 and Table 5.2) and *behavioral constructs* (nine lower rows in Table 5.1). We explain these constructs as follows.

5.1.1 Structural constructs

This section explains the programming constructs corresponding to the concepts used for modeling the component-based architecture structure, including *Port* and *Binding*.

Port A UML port does not have an equivalent element in code. In the extended language, we propose programming constructs based on template mechanisms of the standard language corresponding to UML ports. *RequiredPort*<*T*> and *ProvidedPort*<*T*> (see Table 5.1) are equivalent to UML uni-directional ports, which have only one required or one

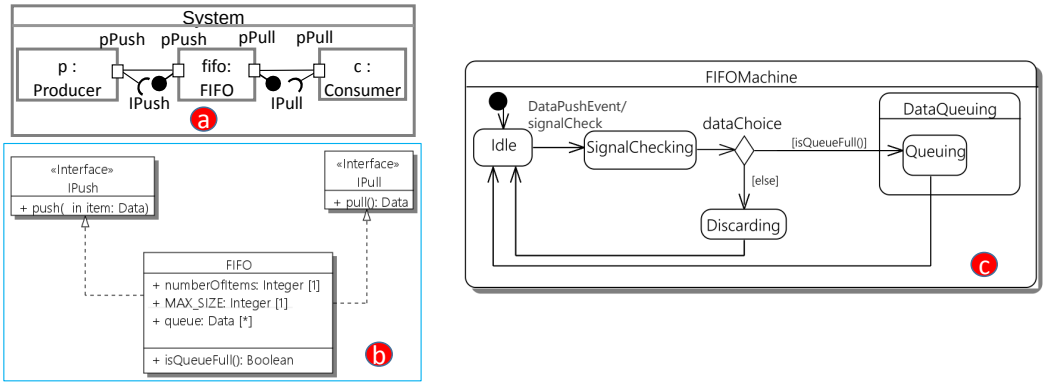


Figure 5.1: Architecture and behavior example model

```
1. class System {
2. public:
3.   Producer p;
4.   Consumer c;
5.   FIFO fifo;
6.   void configuration(){
7.     bindPorts(p.pPush, fifo.pPush);
8.     bindPorts(c.pPull, fifo.pPull);
9.   }
10.}
11.class IPull {
12.public:  virtual Data* pull() = 0;
13.}
14.class IPush {
15.public:
16.  virtual void push(Data& data) = 0;
17.}
18.class Producer {
19.public:  RequiredPort<IPush> pPush;
20.};
21.class Consumer {
22.public:  RequiredPort<IPull> pPull;
23.};
24.class FIFO : public IPush, IPull {
25.public:
26.  ProvidedPort<IPush> pPush;
27.  ProvidedPort<IPull> pPull;
28.  Data* pull(){//fine-grained code}
29.  void push(Data& data){//..}
30.  //attributes + methods…
```

```
31.Statemachine FIFOMachine {
32.  InitialState Idle{};
33.  State SignalChecking {
34.    StateEntry entryCheck();
35.    StateExit exitCheck();
36.  };
37.  State DataQueuing {
38.    StateEntry entryQueue();
39.    State Queuing();
40.  };
41.  State Discarding{};
42.  PseudoChoice dataChoice{};
43.  CallEvent(push(Data&)) DataPushEvent{};
44.  TransitionTable {
45.    ExT(Idle,SignalChecking,
46.        DataPushEvent,NULL,signalCheck);
47.    ExT(SignalChecking,dataChoice,
48.                NULL,NULL,NULL);
49.    ExT(dataChoice,Queuing,NULL,valid,NULL)
50.  }
51.};
52.void entryCheck(){//fine-grained code}
53.void exitCheck(){//fine-grained code}
54.void entryError(){//fine-grained code}
55.void signalCheck(Data& item) {
56.//trans effect from Idle to SignalChecking
57.}
58.bool valid(){return !isQueueFull()}
59.}
```

Figure 5.2: Generated extended code example

Table 5.1: Mapping between UML and Extended Language and Examples-1

UML	Extended Language	Code example in Fig. 5.2
Port requiring an interface *I*	Attribute typed by *RequiredPort*<I>	Ports *pPush* and *pPull* at lines 19 and 22
Port providing an interface *I*	Attribute typed by *ProvidedPort*<I>	Ports *pPush* and *pPull* at lines 26-27
Bidirectional port providing *R* and *P*	*BidirectionalPort*<*R,P*>	Not available in the example
Connector	Binding	Lines 7-8
State Machine	*StateMachine*	The FIFO state machine at lines 31-51
State	*State/InitialState*	State *SignalChecking* at lines 33-36
Region	*Region*	Not available in the example
Pseudo state	Class-like elements	The *dataChoice* pseudo state at line 42
Action/Effect	Method	Methods at lines 52-57
Transitions	Transition table	Transition table at lines 44-50
Event	Event	The call-event at line 43
Deferred Event	State attribute typed by deferred event type	Not available in the example
Transition guard	Method returning a boolean	The *valid* method at lines 58-59 as the transition guard at line 49

provided interface. The *T* template parameter is an interface in code (e.g. *interface* in Java or class with *pure virtual* methods in C++) equivalent to the interface required/provided by a UML port. *BidirectionalPort*<*R,P*> is also proposed to map to UML bidirectional ports, which have one *R* required and one *P* provided interface. Because a UML port is translated into an attribute in the extended code, a UML port with multiplicity > 1 can be transformed into an array attribute with its size as the multiplicity of the port. Since the size of an array should be fixed, we only allow fixed multiplicities for model elements to avoid runtime memory allocations of component parts and ports in embedded systems and to specify the exact number of components and inter-component connections at design-time. This restriction is similar to that of the approach in [Ciccozzi 2014].

Lines 19 and 22 in Fig. 5.2 on page 67 show ports with a required interface and lines 26-37 show ports with a provided interface of the *Producer*, *Consumer*, and *FIFO* classes respectively.

Binding A binding (see Table 5.1, row 4) connects ports on two parts. It is equivalent to a UML connector that can connect two UML ports. Here we consider the case where a UML connector has two connector ends. A method call to our predefined method *bindPorts* connects two ports. This method can be called for various types of connectors. Table C.1 on page 133 shows the equivalences of different connector types and *bindPorts*. A call of *bindPorts* takes as input four parameters: the first two are port references; and the last two are the multiplicities of the two connector ends of the connector. The multiplicities of

the connector ends are used for specifying which *connector pattern/topology* or *connection pattern/topology* used for interactions between component instances in the architecture. There are two types of connector patterns: *array* and *star*. The last two parameters are by default 1's to simplify the array pattern where the multiplicities of the connector ends are 1's. The connector patterns are defined by the Precise Semantics for UML Composite Structure (PSCS) [OMG 2015] standard of OMG. We briefly describe these two patterns and how to relate to the proposed constructs in the followings.

Precise Semantics for UML Composite Structure (PSCS) PSCS defines how precisely the considered composite structure elements, part, port and connector in particular, should behave in presence of multiplicities, that are not well supported in the literature. For parts, ports, and connectors, PSCS provides two patterns with conditions constraining the multiplicities of these elements. The two patterns are called *array pattern* and *star pattern*.

To give a clear explanation of these patterns, assuming that two parts A and B with their p_A and p_B respective ports are connected by a connector with two ends e_A and e_B; $mul(e)$ is the multiplicity of the e element, e.g. we say that there are $mul(A)$ instances of the A part.

- **Array pattern** [OMG 2015]: In an array pattern, each port instance at one end of the connector, e.g. p_A, is only connected to another port instance at the other end of the connector, e.g. p_B. The conditions for the multiplicities for the elements are as belows:

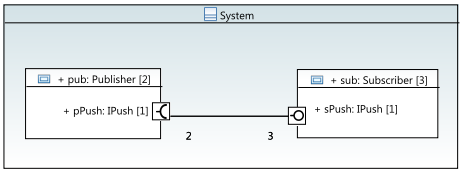
$$mul(e_A) = mul(e_B) = 1 \tag{5.1}$$
$$mul(A) * mul(p_A) = mul(B) * mul(p_B) \tag{5.2}$$

  The connectors in the producer-consumer example in Fig. 5.1 on page 67 are of the array pattern in which the multiplicities of all of the parts, the ports, and the connector ends are 1. The multiplicities of these elements satisfy the 5.1 and 5.2 conditions.

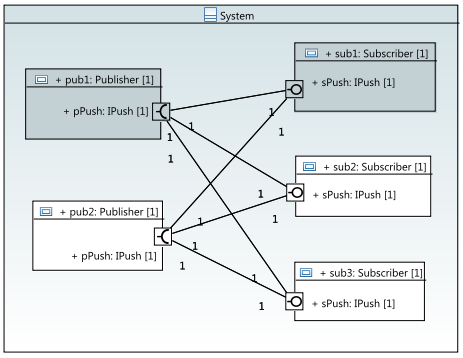
- **Star pattern** [OMG 2015]: In a star pattern, each port instance at one end of the connector, e.g. an instance of p_A, is connected to all port instances at the other end of the connector, e.g. $mul(B) * mul(p_B)$ instances of p_B. Requests from an instance of p_A to p_B is propagated alternatively to instances of p_B. The conditions for the multiplicities for the elements are as belows:

$$mul(A) * mul(p_A) = mul(e_A) \tag{5.3}$$
$$mul(B) * mul(p_B) = mul(e_B) \tag{5.4}$$

  To illustrate this star pattern, let's consider the publisher-subscriber example in Fig. 5.3a. The system has a *pub* publisher part with 2 instances ($mul(pub) = 2$) and a *sub* subscriber part with 3 instances ($mul(sub) = 3$). The multiplicities of the connector ends of the connector between the two parts satisfy the 5.3 and 5.4 conditions for a star pattern. This example is equivalent to the composite structure in Fig. 5.3b, where



(a) Composite structure diagram for a publisher-consumer example with multiplicities



(b) Composite structure diagram for a star pattern

Figure 5.3: Composite structure diagram for a publisher-consumer example with multiplicities for a star pattern connector

the *pPush* port of a *pub* part instance is connected to all of the *sPush* port instances of all *sub* instances. The *pPush* ports of *pub1* and *pub2* become *sending/forwarding points* that allow to alternatively send out data to its connector-connected subscribers instances. For example, the first request from the first instance of *pPush* is sent to the first instance of *sub* and then the second request to the second instance of *sub*.

These pattern conditions are also applicable to delegation connectors.

Table C.1 on page 133 shows a specification of how to map from UML connectors with multiplicities for connector ends and ports to invocations of *bindPorts* and parameters.

For example, lines 7-8 in Fig. 5.2 on page 67 show two invocations of *bindPorts* for two port-port connectors of array pattern. Each of the invocation takes as input two ports (the two ports of the producer and the fifo, for example) and the multiplicities of the connector ends for the array pattern are by default 1s. Because, in this case, the UML connectors between the parts of the system in Fig. 5.1 are of the array pattern. Each code class associated with a UML component contains a single configuration (as a method in lines 6-9) with bindings. The configuration method is restricted to have only invocations of *bindPorts* for synchronization ease. Statements other than invocations of *bindPorts* in the configuration method do not have any effect.

Other elements in the UML class diagram in Fig. 5.1 are mapped to the corresponding code elements as found in industrial tools such as IBM Rhapsody [IBM 2016a] and Enterprise Architect [SparxSysems 2016]. UML parts of a class, e.g. *p, fifo, c*, are mapped to composite attributes of the corresponding class at the code level, e.g. the *System* class; the UML operations and properties are mapped to the class methods and attributes, respectively; the UML interfaces (*IPush* and *IPull*) mapped to interfaces in code, e.g. classes with pure virtual methods in C++ at lines 11-17 in Fig. 5.1.

Usage of additional constructs The purpose of adding new programming constructs is to (1) provide bidirectional mapping between model and code and (2) allow programmers to write fine-grained behavior of components using the constructs: programmers can use the required and/or provided interface of a port to call the methods of the interface. To do it, we provide attribute interfaces as members of the port additional constructs corresponding

Table 5.2: Mapping between UML and Examples of Extended Language-2

UML and MARTE	Extended Language	Example in Listing 5.1
In flow port	Attribute typed by *InFlowPort<Sig>*	Ports *pInData* at line 10
Out flow port	Attribute typed by *OutFlowPort<Sig>*	Ports *pOutData* at lines 3 and 11
Bidirectional flow port	Attribute typed by *InOutFlowPort<Sig>*	Not available in the example
UML Signal	A class	The example does not show a signal class

to the UML ports as follow: a *requiredIntf* attribute (a class attribute in Java or a pointer attribute in C++, e.g.) typed by the required interface of the required or bidirectional port and a *providedIntf* attribute typed by the provided interface of the provided or bidirectional port. For example, to call the *push* method implemented by the *fifo* from the producer, a programmer can write *pPush.requiredIntf->push(data)* in fine-grained code of the producer.

Flow port UML only provides service ports that have provided and/or required interfaces. Some UML extensions/profiles such as MARTE [OMG 2011] define *flow ports* to support data-flow like communication schemas in additions. Flow ports enable message-driven and data-flow oriented communication between components, where messages exchanged between ports represent data items. The direction of a data-flow of a flow port can be *in/out/inout*. The data items exchanged between components through flow ports are modeled in terms of UML signals. Papyrus-RT [Posse 2015] also allows the exchange of messages between ports. The difference is that Papyrus-RT uses the concept of a *protocol* to specify which messages are incoming to or outgoing from a port of a component [Papyrus-RT 2017]. Moreover, a *message* in Papyrus-RT protocols is represented as an *operation*, which can have data as its parameters, while we use the concept of UML Signal with the MARTE profile stereotypes to better represent a message-oriented or data flow-oriented communication. Furthermore, a flow port is useful when being used with signal-events for a UML-SM. The details of signal-events are discussed in Subsection 5.1.2.

In the bidirectional mapping, new programming constructs are also proposed correspondingly to the UML flow ports. Table 5.2 shows our mapping and examples for flow ports. Similarly to the service ports, there are also three template-based flow port programming constructs including *InFlowPort*, *OutFlowPort*, and *InOutFlowPort* with the template parameter as a class in the source code, transformed from the UML Signal of the corresponding UML flow port. The multiplicities of flow ports are translated into code exactly similar to that of service ports as previously described.

For better understanding of flow ports, let's redesign the producer-consumer example by using flow ports. The data items flow from the producer to the *fifo* (through the connector between the *p* and *fifo* parts) and then to the consumer. The changes made to the design in Fig. 5.1 (a) on page 67 are as follows:

- *pPush* and *pPull* of the producer and the fifo become *OutFlowPort* ports, namely, *pOutData* in their respective class since data are sent from these ports.
- *pPush* and *pPull* of the fifo and the consumer become *InFlowPort* ports, namely,

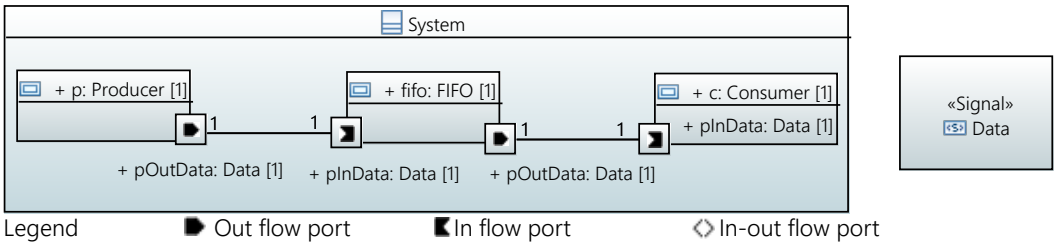


Figure 5.4: Composite structure diagram of the producer-consumer example using flow ports

pInData in the respective class since data are received on these ports.

Fig. 5.4 shows the composite structure diagram of the flow port-based producer-consumer example with the use of the MARTE flow port stereotypes. The data flow between the flow ports is visible through the use of the MARTE stereotype notations.

Listing 5.1 shows the generated code for the flow port-based producer-consumer example. The constructs *OutFlowPort* and *InFlowPort* are used. Note that, in this example, the *FIFO* class no longer implements the *IPush* and *IPull* interface for receiving signal instances since the signal reception is now handled by the state machine of *FIFO* (not shown in Listing 5.1 for simplification). The details of how the reception is handled by the state machine are presented in Subsection 5.1.2 and Section 5.4.

From an implementation perspective, the producer sends data items to the *fifo* via the port *pOutData* by calling *pOutData.outf->push(item)* (lines 4-6). Similarly to the service ports, out-flow ports also have an interface attribute *outIntf* for writing code to send signals from a component to another component and in-flow ports have an interface attribute *inIntf*. If the behavior of the receiving component (the *fifo*, for example) is described by a state machine, the *push* method will fire a signal-event instance that is handled by the state machine later. The details of signal-events are discussed in sub-section 5.1.2.

Listing 5.1: Producer-consumer using flow ports

```
class Producer {
public:
  OutFlowPort<Data> pOutData;
  void sendToFifo(Data& item) {
    pOutData.outIntf->push(item);
  }
}
class FIFO {
public:
  InFlowPort<Data> pInData;
  OutFlowPort<Data> pOutData;
}
class Consumer {
public:
  InFlowPort<Data> pInData;
}
class System {
public:
  Producer p;
  FIFO fifo;
  Consumer c;
  void configuration() {
    bindPorts(p.pOutData, fifo.pInData);
    bindPorts(fifo.pOutData, c.pInData);
  }
}
```

In the next sub-section, we present our proposed additional programming constructs corresponding to the behavioral modeling concepts, in particular UML-SM elements.

```
31.Statemachine FIFOMachine {
32.  InitialState Idle{};
33.  State SignalChecking {
34.    StateEntry entryCheck();
35.    StateExit exitCheck();
36.  };
37.  State DataQueuing {
38.    StateEntry entryQueue();
39.    State Queuing{};
40.  };
41.  State Discarding{};
42.  PseudoChoice dataChoice{};
43.  CallEvent(push(Data&)) DataPushEvent;
44.  TransitionTable {
45.    ExT(Idle,SignalChecking,
46.        DataPushEvent,NULL,signalCheck);
47.    ExT(SignalChecking,dataChoice,
48.                  NULL,NULL,NULL);
49.    ExT(dataChoice,Queuing,NULL,valid,NULL)
50.  }
51.};
52.void entryCheck(){//fine-grained code}
53.void exitCheck(){//fine-grained code}
54.void entryError(){//fine-grained code}
55.void signalCheck(Data& item) {
56.//trans effect from Idle to SignalChecking
57.}
58.bool valid(){return !isQueueFull()}
59.}
```

Figure 5.5: Generated extended code for FIFO state machine example copied from Fig. 5.2

5.1.2 Behavioral constructs

In our approach, UML-SMs are used for modeling the discrete event-driven behavior of components. Our behavioral programming constructs correspond to the UML-SM concepts at the modeling level. These behavioral constructs are grouped into three parts: topology, events, and transition table in the extended code.

Topology A topology contains the constructs to describe the state machine hierarchy. The root of the topology is specified via the *StateMachine* as in Fig. 5.5. Note that, for readability reasons, Fig. 5.5 is coped from Fig. 5.2 on page 67. A *StateMachine* contains class-like declarations of vertexes (see Fig. 5.6 for more details about the syntax). Similarly to the concepts of vertexes in UML-SMs, a vertex can be either a state, which can in turn contain one or more regions, or a pseudo state. The vertex elements are declared as state machine sub-elements.

In UML, a UML-SM state can have associated actions: *entry*, *exit*, and *doActivity*. In the mapping, the *entry/exit/doActivity* state actions are declared within the corresponding state in the extended code as state methods-like. And, these actions must be implemented as methods in the owning class and have no parameter.

For example, in Fig. 5.5, *Idle* is declared as an initial state. The *SignalChecking* state (lines 33-36) is declared with its entry and exit state actions, *entryCheck* and *exitCheck*, respectively. The owning class *FIFO* of the state machine implements the two methods *entryCheck* and *exitCheck* (lines 52-53) for the state actions. The separation of declarations and definitions of state actions is similar to that of C++ where declarations are in header files and definitions in source files.

⟨statemachine⟩ ⊨ StateMachine ⟨name⟩ { ⟨vertexes⟩ ⟨events⟩ ⟨trans-table⟩} ;
⟨vertexes⟩ ⊨ ⟨vertex⟩ ; ⟨vertexes⟩ | ⟨vertex⟩
⟨vertex⟩ ⊨ ⟨state⟩ | ⟨initial-state⟩ | ⟨pseudo-state⟩
⟨state⟩ ⊨ State ⟨name⟩ { ⟨deffered-events⟩ ⟨state-content⟩ } ;
⟨initial-state⟩ ⊨ InitialState ⟨name⟩ { ⟨initial-effect⟩ | λ ⟨deffered-events⟩ ⟨state-content⟩ } ;
⟨state-content⟩ ⊨ ⟨state-actions⟩ ⟨connection-points⟩ ⟨vertexes⟩ | ⟨regions⟩
⟨initial-effect⟩ ⊨ InitialEffect ⟨name⟩ ();
⟨state-actions⟩ ⊨ ⟨state-entry⟩ ⟨state-exit⟩ ⟨state-doactivity⟩
⟨state-entry⟩ ⊨ StateEntry ⟨name⟩ ();
⟨state-exit⟩ ⊨ StateExit ⟨name⟩ ();
⟨state-doactivity⟩ ⊨ StateDoActivity ⟨name⟩ ();
⟨connection-points⟩ ⊨ ⟨entry-point⟩ | exit-point | λ ⟨connection-points⟩
⟨entry-point⟩ ⊨ PseudoEntryPoint ⟨name⟩{};
⟨exit-point⟩ ⊨ PseudoExitPoint ⟨name⟩{};
⟨pseudo-state⟩ ⊨ ⟨pseudo-keyword⟩ ⟨name⟩{};
⟨pseudo-keyword⟩ ⊨ PseudoInitial | PseudoTerminate . . . | PseudoDeepHistory
⟨regions⟩ ⊨ ⟨region⟩⟨regions⟩ | λ
⟨region⟩ ⊨ Region ⟨name⟩{⟨vertexes⟩};
⟨events⟩ ⊨ ⟨event⟩ ⟨events⟩ | λ
⟨event⟩ ⊨ ⟨Call-event⟩ | ⟨Time-event⟩ | ⟨Signal-event⟩ | ⟨Change-event⟩
⟨name⟩ ⊨ *Identifier*
⟨deferred-events⟩ ⊨ ⟨deferred-event⟩⟨deferred-events⟩ | λ
⟨deferred-event⟩ ⊨ DeferredEvent ⟨name⟩;

Figure 5.6: A concrete syntax for state machine topology in C++. For syntax of event types and transitions, see Fig. 5.7 and 5.8 for more details. λ denotes an empty string

Note that the state machine in the extended code is syntactically valid in the standard programming language by using its built-in features such as macros in C++ and annotations in Java. Here, we do not impose a specific syntax for the state machine in the extended code. It is left for the developers of the synchronization mechanism to implement a concrete syntax. The constraints for the syntax are: (1) it must be syntactically valid in the standard programming language by using built-in features; and (2) it must explicitly explore the UML-SM concepts within the code. For example, the extended code state machine hierarchy can be defined as an object-oriented class hierarchy, in which the *StateMachine* keyword can be defined as a macro for replacing the *class* keyword. It means that the underlying layer behind the macros for state machine elements is class hierarchy. That is why it is compilable with standard compilers. Therefore, programmers can easily write fine-grained code for the state action methods within a standard IDE while fully profiting from assistance features of the IDE such as automatic completion.

Concurrent states can have multiple orthogonal regions in the extended code. A region in turn can have multiple vertexes defined. Listing 5.2 demonstrates an example of a component *A* with a state machine *SM* as its behavior in the C++ extended code. The state machine has an initial concurrent state with two regions, namely *OrthogonalRegion1* and *OrthogonalRegion2*. While the representation of states and regions is very similar to Textual Modeling Language (TML) [Mazanec 2012], the difference is that the state machine in the extended code is syntactically valid in the standard programming language.

Listing 5.2: Example of orthogonal regions

```
1 class A {
    StateMachine SM {
3     InitialState S1 {
        Region OrthogonalRegion1 {
5         InitialState S2{};
        }
7       Region OrthogonalRegion2 {
          InitialState S3{};
9       }
      }
11  }
  }
```

Pseudo states can be declared within *Statemachine/State/Region* in the extended code. The syntax is similar to class declarations with keywords different from *class*. The keyword of pseudo states is one of *PseudoEntryPoint, PseudoExitPoint, PseudoInitial, PseudJoin, PseudoFork, PseudoChoice, PseudoJunction, PseudoShallowHistory, PseudoDeepHistory, PseudoTerminate*, which correspond to the pseudo states defined in UML-SM. For example, line 42 in Fig. 5.5 declares the *dataChoice* choice pseudo state corresponding to the pseudo state in the *FIFOMachine* state machine at the model level.

Events In UML, four event types including *CallEvent*, *SignalEvent*, *TimeEvent*, and *ChangeEvent* are defined for modeling discrete event-driven behavior using UML-SMs. The semantics of these events is clearly defined in the UML specification and is briefly described in the list below.

- **Call-event**: A call-event is associated with an operation/method and emitted if the operation is invoked.
- **Signal-event**: A signal-event is associated with a UML signal type containing data. When a component, whose behavior is described by a UML state machine, receives a message/signal through its *in/inout* flow ports, a signal-event is automatically emitted

⟨Call-event⟩ ⊨ CallEvent (⟨op_signature⟩) ⟨name⟩{} ;
⟨Time-event⟩ ⊨ TimeEvent (⟨dur⟩) ⟨name⟩{} ;
⟨Signal-event⟩ ⊨ SignalEvent (⟨sig_name⟩) ⟨name⟩{} ;
⟨Change-event⟩ ⊨ ChangeEvent (⟨change_expr⟩) ⟨name⟩{} ;

Figure 5.7: A concrete syntax for event declarations in C++

and stored in an event queue for later processing by the state machine. Note that, in case there is a delegation connector from the port of the component to one of its inner parts, a signal-event instance is emitted to the inner part.

- **Time-event**: A time-event specifies a wait period, starting from the time when a state with an outgoing transition triggered by the time-event is entered. The time-event is emitted if the state remains active longer than the wait period. Once emitted, it triggers the transition. In other words, the state, which is the source vertex of a transition triggered by a time-event, will remain active for a maximal amount of wait period specified by the time-event.

- **ChangeEvent**: A change-event has a boolean (change) expression and is fired if the expression value changes from false to true. For example, a change-event can be used to detect some information such as changes of temperature measured by a sensor. The change expression is, in the mapping, transformed into a change method, which returns a boolean value.

Following the UML specification [Specification 2015], the processing of a call-event can be either *synchronous* or *asynchronous* depending on the design of the operation associated with the call-event. Synchronous processing of a call-event means that it runs within the thread of the operation caller. Developers should pay attention to the use of synchronous call-events to avoid a *deadlock* problem that will be presented in detail later in Subsection 5.4.3.3 on page 98. The processing of other events is asynchronous meaning that the received events are stored in an event queue which is maintained by the component of the state machine at runtime for later processing. A simple usage of synchronous call-events is in user interaction (UI) applications, in which users click on a button. The click then emits an event and calls user code for synchronously processing the event to respond to the users. We support all of these events with the aforementioned semantics.

Again, we do not impose a specific syntax for event declarations. As a proof of concept, Fig. 5.7 gives a concrete syntax example for the syntax of the event declarations in C++.

Essentially, each component in the syntax carries known semantics defined in the UML specification which is described as follows:

- `name`: The unique identifier for an event.

- `op_signature`: The signature of the operation/method associated with a call-event. This signature includes the name and the parameter types of the method.

- `dur`: The duration in millisecond associated with a time-event.

- `sig_name`: The name of the signal class type (a UML Signal is transformed into an object-oriented class) associated with a signal-event. Typically, upon reception of an

instance of this signal type on a port, a signal-event is emitted. Listing 5.3 shows an example of a signal-event *SE_Example* for the *Data* signal.

- `change_expr`: The name of the change expression method associated with the change-event.

Transition table It describes the mapping of our syntactical constructs equivalently to UML transitions at the model level. Three kinds of UML transitions, *external*, *local*, and *internal* are supported. The differences between the transition kinds are previously described in 2.1. While an external or a local transition is specified as `(kind, source, target, guard, event-name, effect)`, an internal transition, which does not need to know its target vertex because its source and target are the same, is specified as `(kind, source, guard, event-name, effect)` with components defined as follows:

- `kind`: transition kind (external, local, internal).
- `source`: The name of the source vertex of the transition.
- `target`: The name of the target vertex of the transition.
- `event-name`: The name of the event that triggers the transition.
- `guard`: The name of a transition guard method. A guard method in the extended code is equivalent to a transition guard at the model level. A guard method is a member method of the component, whose behavior is described by the state machine, and returns a boolean value. The guard method allows programmers to write the boolean expression of the UML guard in a method-like style, which programmers are familiar with. If the transition is triggered by an event, the guard method should have access to the data of the event for reasoning the activation of the transition. The guard method, therefore, can have parameters depending on the event type. If the event is a call-event, the guard method has the same parameters as the associated method of the call-event has. The values that will be passed to the parameters of the guard method are those of an invocation of the method. If the event is a signal-event, the guard method has a parameter typed by the signal type.
- `effect`: The name of a transition effect method. An effect method in the extended code is equivalent to a transition effect at the model level. It is executed if the returned value of the guard method is true; Similarly to the guard method, the effect method also has parameters that follow the same rules of the parameters of the guard method.

As a proof of concept, we give the syntax for declaring transition table and transitions as shown in Fig. 5.8.

Lines 44-50 in Fig. 5.5 show a transition table with three external transitions (defined by the "ExT"keyword). The first one is from *Idle* to *SignalChecking*, triggered by the *DataPushEvent* call-event declared within the state machine, and has *signalCheck* as its transition effect. The call-event is declared at line 50 and an instance of it is emitted whenever there is an invocation of the *push* method of the *FIFO* class. The processing of the emitted event activates the transition from *Idle* to *SignalChecking*, and executes the *signalChecking* transition effect method. The data item brought by the invocation will be checked for validity and further put to the queue or discarded. The *signalCheck* method is implemented within the *FIFO* class owning the state machine, and has the same formal

⟨trans-table⟩ ⊨ ⟨transitions⟩
⟨transitions⟩ ⊨ ⟨transition⟩⟨transitions⟩ | λ
⟨transition⟩ ⊨ ⟨ExternalTrans⟩ | ⟨LocalTrans⟩ | ⟨InternalTrans⟩
⟨ExternalTrans⟩ ⊨ ExT(⟨source⟩,⟨target⟩,⟨event-name⟩,⟨guard⟩,⟨effect⟩);
⟨LocalTrans⟩ ⊨ ExT(⟨source⟩,⟨target⟩,⟨event-name⟩,⟨guard⟩,⟨effect⟩);
⟨InternalTrans⟩ ⊨ ExT(⟨source⟩,⟨event-name⟩,⟨guard⟩,⟨effect⟩);

Figure 5.8: A concrete syntax for transitions

parameters as the *push* method associated with the call-event. This first transition does not have a guard method since "NULL" is specified for this guard.

The second transition does not have any guard, triggering event, and effect. The last transition has a guard method *valid* that is implemented in the *FIFO* class.

Note that the state machine example in Fig. 5.1 and 5.5 can replace the *DataPushEvent* call-event of the transition from *Idle* to *SignalChecking* with a signal-event associated with the *Data* signal.

Deferred event A state can declare *deferred events* by introducing attributes typed by our class-like additional construct *DeferredEvent* and named as event names to be deferred. A deferred event will not be processed while the state remains active. The deference of events is used to postpone the processing of some low-priority events while the state machine is in a certain state. The execution semantics of deferred events is: given a current active state with declared deferred events and the event queue, the deferred events appearing at the head of the queue will be moved to a deferred set and pushed back to the front of the queue once a non-deferred event is processed. Listing 5.3 shows a state machine with an initial state S1 that defers the *SE_ Example* signal-event.

Listing 5.3: Example of signal-event and deferred event

```
class A {
  StateMachine SM {
    InitialState S1 {
      DeferredEvent SE_Example;
    }
    SignalEvent(Data) SE_Example;
  }
}
```

We have shown a bidirectional mapping between the architecture model and the extended code. The next sections show the text-to-text transformation to make the additional constructs in the extended code executable and debug-able.

5.2 Overview of in-place text-to-text transformation/Preprocessor

As a reminder, the previously generated extended code contains our template-based and syntactic additional programming constructs and user fine-grained code, e.g. state machine action methods. This extended code itself is compilable by standard compilers (because the additional constructs are created by using built-in features of the standard programming language) but natively not executable and debug-able (see Chapter 3 on page 39 for

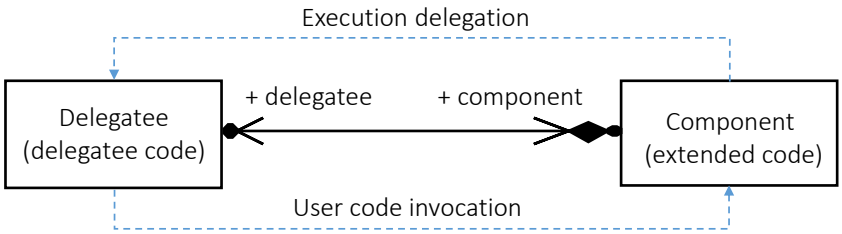


Figure 5.9: Delegation from a component to its delegatee

explanation). That is why an *in-place text-to-text (T2T) transformation, which acts as a preprocessing* is needed. The role of the T2T transformation is similar to that of a Java annotation processing as previously described in Chapter 3 on page 39.

For each component class containing the additional constructs such as ports, bindings, and state machine elements, the T2T transformation creates an additional code file/class, whose name has *Delegatee* as its postfix, complementing the extended code. In the followings, we refer "extended code component" as a component within the extended code, which use additional constructs, and "delegatee" as the additional code class created by the T2T transformation. The created delegatee classes/files are in the same repository with the extended code but hidden from developer perspectives and not intended to be modified by the developers. These delegatee files play as role of *dynamic library code* executing the runtime semantics and logic specified by the additional constructs. If programmers modify the extended code in a file, the T2T transformation will re-generate the dynamic library code file so that this latter executes the desired semantics of the modified extended code.

Fig. 5.9 shows the relationships/associations and control flows between an extended code component and its corresponding generated delegatee as follows:

- An extended code component owns a composite attribute typed by its delegatee class.
- The execution of the extended code component is passed to the delegatee through the owned composite attribute.
- The user code (e.g., state entry/exit actions or transition effects) in the extended code component is invoked by the delegatee execution via a *component* reference to the extended code component.

The process of generating delegatee code from the additional constructs is similar to a code generation process from component-based design and state machines. This code generation is supported by several tools such as IBM Rhapsody [IBM 2016a], Papyrus-RT [Posse 2015], and Enterprise Architect [SparxSysems 2016]. However, the code generated from these tools is mixed with fine-grained behavior code such as state action or transition effect code, which makes the reconstruction of state machines and component-based elements from code impossible. Our code generation, on the other hand, completely separates the delegatee code and user-code in different files. By this way, the code after the T2T transformation is composed of two parts: *visible files*, which are *extended code files* and modifiable by programmers, and *delegatee code files*, which are automatically generated or modified by the T2T transformation if the extended code files change. The delegatee code after the transformation is compilable, executable, and debug-able (but not modifiable in the sense that modifications in the delegatee code are overwritten by the transformation).

Example formalization To formally illustrate the delegation from an extended code component to the corresponding delegatee, Fig. 5.10 shows a sequence diagram that formalizes the interactions between the extended code component and the delegatee during delegation for the producer-consumer example. For simplification, let's assume that the main method of the program creates an instance of the system. The creation of the system instance entails the creation of its corresponding delegatee instance. The three parts *p*, *fifo* and *c* of the system are then created by the system instance. For each created part, an instance of the delegatee associated with the extended code component of the part is instantiated. The delegation from the system instance to the system delegatee is that: to establish the connections between ports communicated by connectors at the model level, the main method calls *Create Connections* of the system delegatee instead of calling methods of the extended code system. The details of the connections through the system delegatee are presented in Section 5.3. Similarly, the delegation of the state machine behavior execution of the *FIFO* instance is the call of *Start State Machine Behavior* of the *fifo* delegatee. Hence, the state machine execution of the *fifo* is run within the fifo delegatee instance. More information about such execution delegation is shown in Section 5.4.

The idea of the complete separation of the delegatee code from the extended code is, to some extent, similar to that of *deep separation* presented in [Zheng 2012]. *Deep separation* intends to support the co-evolution of architecture model, specified by the xADL 2.0 architecture description language with behavior support specified by simplified state machine and sequence diagrams, and Java implementation, by separating architecture-prescribed code from user-defined code. However, the separation in this thesis differs from *deep separation* fundamentally: *Deep separation* does not allow users to (1) change architecture at the code level and (2) modify the architecture and implementation concurrently. The separation in this thesis, on the other hand, is the way of making the proposed additional constructs executable and allows both of changing architecture at the code level and modifying architecture model and code concurrently. Furthermore, the support of state machines in this thesis is more rigorous than that of the deep separation: we support a complete set of UML-SM elements rather than a simple set of the elements as in the deep separation approach. In addition, this thesis intends to generate code that is conforming to the UML specification while the deep separation in [Zheng 2012] follows the xADL [Khare 2001] architecture description language.

The T2T transformation is based on a set of patterns for generating code from UML-CS and UML-SM. Note that, instead of applying the patterns to generate code within the extended code component as in the existing tools (IBM Rhapshody and Enterprise Architect), the T2T transformation takes as input the information of the constructs declared within the extended code component to generate code for the corresponding delegatee. The details of the patterns are presented in the next sections.

Since the T2T transformation is based on a set of code generation patterns, in the followings, we use the *code generation* and *transformation* terms interchangeably.

5.3 Transformation from Component-Based to Object-Oriented Concepts

This section describes the process and patterns to generate object-oriented code from component-based concepts or additional structural programming constructs contained by the generated extended. A standard compiler can then make use of the extended code

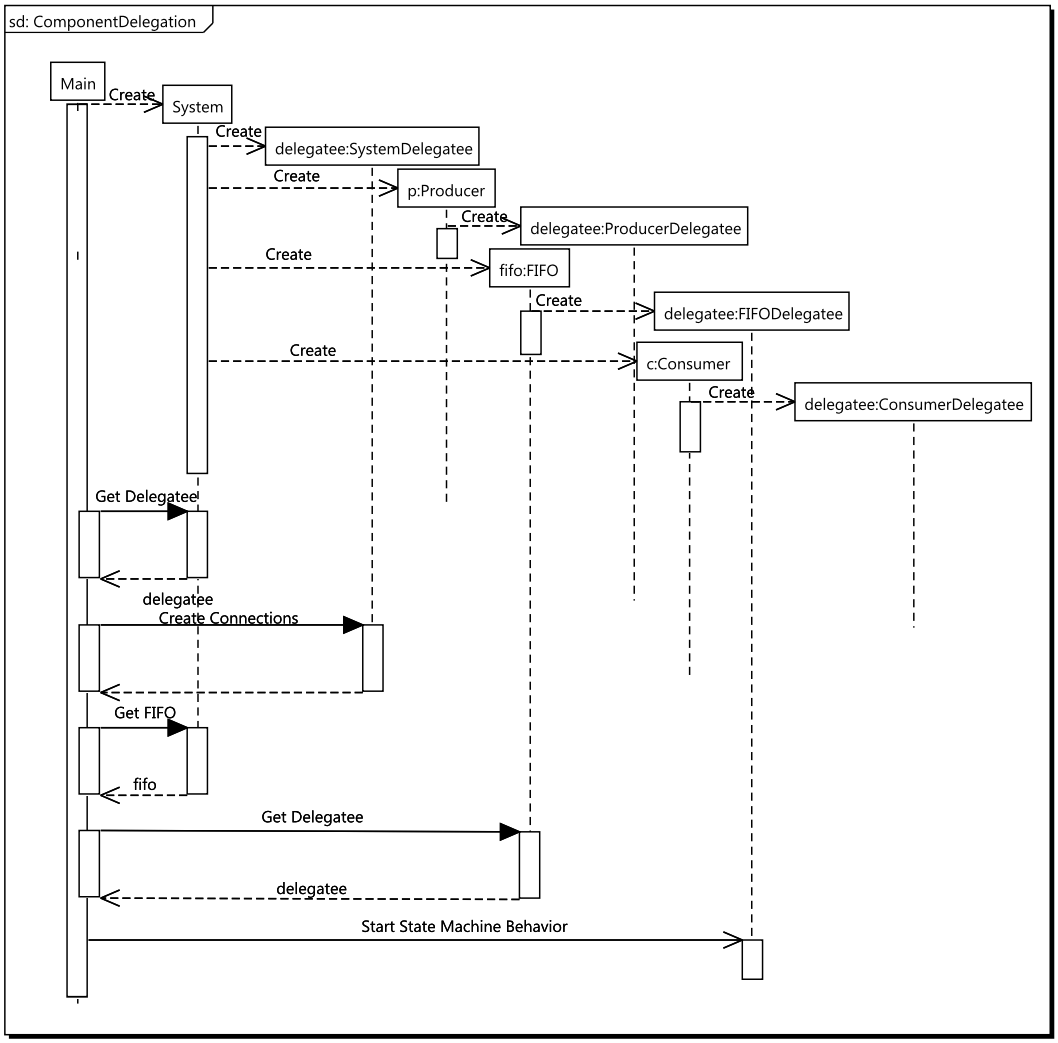


Figure 5.10: Formalization of operation of delegation from extended code to delegatee code for the producer-consumer example when a system is created from the main method

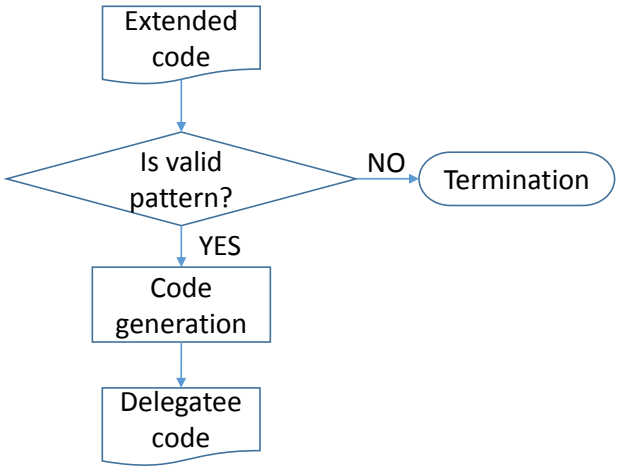


Figure 5.11: Delegation from a component to its delegatee

and the delegatee code to compile, execute and debug them. Because a binding in the extended code is equivalent to a UML connector, in the followings, these two terms are used interchangeably. The objectives are: (1) to generate delegatee code from an extended code component given that the two codes should be physically separated from each other so that programmers do not need to pay attention to the existence of the delegatee code (the extended code is executable from the programmers perspectives); (2) to support the two codes being logically related with each other through delegation as described in Section 5.2 on page 78; and (3) to support a complete set of the structural constructs as well as their semantics, especially the connector patterns, namely the array and star patterns.

Fig. 5.11 shows the overview of the process. The first step is to verify whether the structural constructs contained in the extended code of a component are validated by an array or a star pattern since these patterns have different semantics. The next step is to use a set of code generation patterns to produce delegatee code.

5.3.1 Verification

The verification process checks the multiplicities of structural elements, namely parts, ports, and connector ends for finding out which connection patterns are used in the architecture. There can be multiple connectors from the same port to multiple other ports. PSCS allows such usage of multiple connectors for exploring architectures. In existing code generation tools, they impose a restriction in which a required port is connected to only one provided port [Posse 2015, Ciccozzi 2014]. We allow the use of multiple connectors from the same port and generate delegatee code accordingly. Fig. 5.12 shows examples of using multiple connectors from the same port (pA) to multiple ports (pB and pC in (a)) and a port and a part(pB and c in (b)).

Note that in Fig. 5.12 (b), UML-CS allows a connector to connect a port on a part with a part (without port). We call such a connector a port-part connector that is different from a port-port connector. A port-part connector is useful when using with a legacy object-oriented class without modifying it. For example, in Fig. 5.12 (b), if the class C is a legacy class and does not have any port, we need to add a port to it so that a port-port connector can connect the part a with c. A port-part connector does not need to have this addition

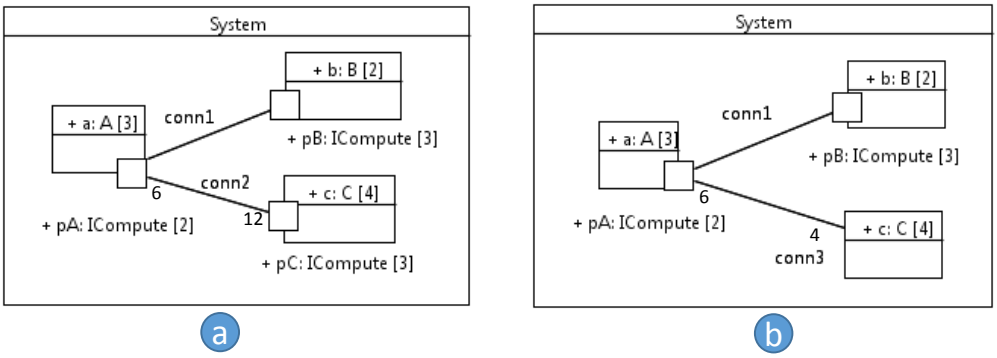


Figure 5.12: Composite structure examples of port-port connectors (a), and both port-port and port-part connectors (b)

and therefore keeps the legacy class intact.

A port-part connector can be considered as a port-port connector in which a virtual port is created in the part and the multiplicity of this virtual port is always 1. For example, in Fig. 5.12 (b), we can assume that the *conn3* connector connects the pA port on the a part to a virtually created port, namely pC, on c where the multiplicity of pC is 1. Every discussion related to port-port connectors can be generalized to port-part connectors, e.g. mapping between a port-part connector and a *bindPorts* method call.

The semantics in Fig. 5.12 (a) is that each port instance of pA connects to:

1. A port instance of pB.
2. All port instances of pC (totally 12 instances since each of 4 part instances of c has 3 port instances).

Therefore, there are actually 13 connections from the pA port during execution. In Fig. 5.12 (a), assume that the pA port requires the *ICompute* interface and pB and pC provide this interface. A call of one of the operations of *ICompute* through pA is propagated to one of 13 instances including: 1 instance of pB and 12 instances of pC. In the PSCS standard, the solution for selecting which instance the call is propagated to is **alternative**. By applying this solution to the example, the first call might be propagated to pB, the second to the first instance of pC, the third to the second instance of pC, and so on. In general, if we consider the 13 instances of pB and pC as an ordered set and pB as the first element (with the index as 0) of the set, the k^{th} call from an instance of pA is propagated to the $(k\%13)^{th}$ instance of the set. In effect, this ability of allowing to use multiple connectors starting from a port greatly simplifies the modeling, abstracts the low-level implementation of the interactions between components, and allows designers to specify connector patterns between components in a condensed way.

In the followings, we discuss the verification of connector patterns for assembly port-port connectors. Assembly port-part connectors are, as previously discussed, similar to port-port connectors. The discussion for delegation connectors, which is very similar to that of assembly connectors, is then presented.

Assembly connectors Let's remind the multiplicity of an element e is written as $mul(e)$. We assume a set of connectors C that start from the same port p. For each connector

$c \in C$, assume that c connects p to the other port p_{other} on a part a_{other} and the connector ends that connect to p and p_{other} are e and e_{other} respectively. The algorithm 1 shows the verification process. The algorithm checks whether each connector $c \in C$ satisfies the conditions for an array pattern or a star pattern that are described in Subsection 5.1.1. This algorithm is actually a generalization of the 5.1, 5.2, 5.3 and 5.4 conditions for checking a single connector satisfying one of the two patterns.

Algorithm 1 Verification for port-port connectors

Require: A set of connectors C that start from the same port p of a part a to other ports
Ensure: Check whether the connectors satisfy the conditions for valid patterns

```
1: procedure VERIFYVALIDPATTERNSOFPORTS(p, a, C)
2:     for ∀c ∈ C do
3:         if mul(e) == 1 ∧ mul(e_other) == 1 then
4:             if mul(p) * mul(a) ≠ mul(p_other) * mul(a_other) then
5:                 Throw exception: Not an array pattern
6:         else
7:             if mul(p) * mul(a) ≠ mul(e) ∨ mul(p_other) * mul(a_other) ≠ mul(e_other) then
8:                 Throw exception: Not a star pattern
```

Delegation connectors In this case, delegation connectors are specializations of assembly connectors that connect a port p of a component to ports of the inner parts of the component. The verification process is similar to Algorithm 1, except that $mul(a) = 1$.

In the next section, the code generation patterns are presented.

5.3.2 In-place text-to-text transformation or code generation pattern for UML-CS elements

This section presents our code generation patterns for UML-CS elements in the extended code, ports and bindings in particular. We start with the patterns for ports. Then, a *createConnections* method is created for each configuration of bindings.

5.3.2.1 Code generation for ports

ProvidedPort For each provided port of an extended code component, a *getter* method is generated within the corresponding delegatee. The purpose of this method is to return the *providedIntf* provided interface attribute of the provided port (see Subsection 5.1.1 on page 66 for more information about this attribute). Obviously, this attribute is initially null. The value assignment of this attribute depends on the internal structure of the class containing the port. If *the multiplicity of the provided port > 1*, the getter has a *portIndex* additional parameter, that indicates which port instance's provided interface should be returned. There are three cases to be considered for provided interface assignment as follows:

1. **Case 1**: If there are no delegation connectors from the provided port as the example in Fig. 5.13 on page 87, the attribute simply refers to the component class (or the extended code component) since this latter must implement the provided interface. Lines 9-12 of Listing 5.5 illustrate this case where the getter method *get_pB* returns

the *component* reference implementing the *ICompute* interface. The significance is: for a request for one of the services provided by a port, unless this port delegates the request to an inner part, the request should be handled directly by the extended code component.

2. **Case 2**: If there are no delegation connectors and there are call-events associated with methods of the provided interface, the delegate must implement the interface. The attribute refers to the delegatee. The implementation of the interface by the delegatee acts as a wrapper and follows the following rules:

   - Rule 1: For a method m of the interface not associated with any call-event, its corresponding wrapper method to be implemented by the delegatee should invoke m of the extended code component. By this way, invocations of the method through the port are also handled by the extended code component similarly to Case 1.
   - Rule 2: For a method m associated with a call-event, the difference between its wrapper method and that of Rule 1 is that its wrapper method calls the event processing method generated from the call-event to tell the state machine execution in the delegatee to handle this event before invoking m of the extended code component. Listing 5.4 on page 86 illustrates this rule. Remember that, in this delegatee code generated from the extended code components of the service port-based producer-consumer example, the *push* method of the *IPush* interface is associated with the *DataPushEvent* call-event. The *FIFODelegatee* class therefore implements *IPush*. The *push* wrapper method calls the *processDataPushEvent* event processing method corresponding to the event to emit an event instance to an event queue (for asynchronous call-events) or synchronously process it (for synchronous call-events).

3. **Case 3**: If there exists at least a delegation connector from the provided port, the initialization of the attribute is done by the *createConnections* method, which is created by the T2T transformation from the configuration of the extended code component. The process of generating code for *createConnections* is described in Subsection 5.3.2.2.

RequiredPort For each required port of a component, a *setter* method is generated within the corresponding delegatee for setting the required interface attribute of the port to an appropriate implementation of the interface. The method has an additional parameter *portIndex* if *the multiplicity of the port* > 1.

InFlowPort An *InFlowPort* port provides an *inIntf* attribute typed by our pre-defined interface *IPush*<*Sig*> as previously described in Section 5.1.1. The template parameter *Sig* indicates the data signal type whose instances flowing through the port. This interface contains a single method, namely *push*. The interface with a signal type here plays the role as the provided interface of the port: other components call the *push* method of the interface to send signal instances. Therefore, the code generation for an in-flow port is similar to that of a provided port.

Listing 5.4: Getters and setters in the delegatee classes of the service-port producer-consumer example

```
class ProducerDelegatee {
public:
  Producer* component;
  void set_pPush(IPush* impl) {
    component->pPush.requiredIntf = impl;
  }
}
class ConsumerDelegatee {
public:
  Consumer* component;
  void set_pPull(IPull* impl) {
    component->pPull.requiredIntf = impl;
  }
};
class FIFODelegatee: public IPush {
public:
  FIFO* component;
  IPush* get_pPush() {
    component->pPush.providedIntf = this;
    return this;
  }
  IPull* get_pPull() {
    component->pPull.providedIntf = component;
    return component;
  }
  void push(Data& data) {
    processDataPushEvent(data);
    component->push(data);
  }

  void processDataPushEvent(Data& data) {
    //method body is generated by state machine code generation process
  }
}
```

OutFlowPort Similar to the discussion of code generation from in-flow ports, an out-flow port acts as a required port with *IPush<Sig>* as required interface. *Sig* is the data signal type flowing out through the port.

BidirectionalPort A bidirectional service port is the combination of a provided port and a required port. The code generation for the bidirectional port is then equivalent to that of the two ports: a getter for the provided port and a setter for the required port.

InOutFlowPort Similar to service ports, the code generation for an in-out flow port is equivalent to that of an in-flow port and an out-flow port.

5.3.2.2 Code generation for connectors/bindings

As shortly described in Section 5.2, a *createConnections* method is generated for each configuration of an extended code component for establishing the connections between ports defined by connectors/bindings, e.g. connect the required interface of a required port to the provided interface of a provided port. This *createConnections* method is created in the corresponding delegatee of the extended code component. In the followings we only detail the code generation pattern for assembly port-port connectors. The discussion for port-port delegation connectors will be shortly presented afterwards.

Code generation for assembly port-port connectors For each required port p of a part a, we assume C as a set of assembly connectors starting from this port. Without loss

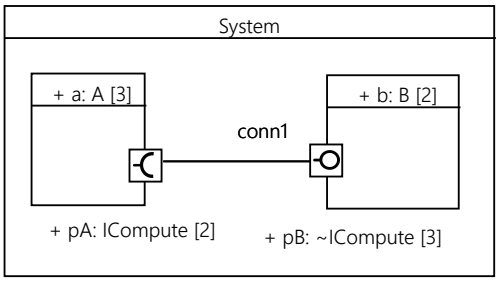


Figure 5.13: Example of single connector for array pattern

of generality, we assume that each port in the extended code has multiple port instances, e.g. port p is an array of $mul(p)$ instances. We distinguish two cases:

- **Single Array**: C contains only one single connector for an array pattern.
- **Hybrid**: C contains connectors for star patterns and connectors for array patterns.

The distinction of the two cases is based on the difference between their semantics. In the first case, following the semantics of the array pattern, an instance of the port p at the one end of the single connector of C is connected to only another one port instance at the other connector end. In contrast, in the second case, an instance of the port p is connected to multiple port instances.

The followings describe the code generation patterns for the two cases.

Single Array This is the simplest case where there is only one connector in C and the multiplicities of the two connector ends are both 1s. Let p and q be the two ports connected by the single connector $c \in C$. Let a and b be the parts with the p and q ports, respectively. Note that, each instance of the a part contains $mul(p)$ instances of the p port. Following the PSCS specification, each instance of p connects to an instance of the q port at the other end of the single connector of C and the 5.2 condition must be held. We denote $a[i].p[j]$ as the j^{th} instance of the p port of the i^{th} instance of the a part. The connections between the instances of p and q are the same as the approach presented in [Ciccozzi 2014]. In this latter, we need to find the $b[k].q[m]$ port that should be connected to $a[i].p[j]$. k and m are computed as follows:

$$k = (i * mul(p) + j)/mul(q) \tag{5.5}$$

$$m = (i * mul(p) + j)\%mul(q) \tag{5.6}$$

Listing 5.5 on page 88 shows the code generation for the example in Fig. 5.13 on page 87. The *createConnections* method uses the equations 5.5 and 5.6 to find out which port of the b part should be connected to a given port of the a part.

Hybrid This case assumes that the set of connectors C containing more than one connector. Let C_{array} and C_{star} be the subsets of C where C_{array} only contains connectors of the array pattern and C_{star} only contains connectors of the star pattern. We have $C = C_{array} \cup C_{star}$. The semantics of this hybrid structure is as follows:

Listing 5.5: Generated code for single array pattern

```
class ADelegatee {
  A* component;
  void set_pA(ICompute* impl, int portIndex) {
    component->pA[portIndex].requiredIntf = impl;
  }
}
class BDelegatee {
  B* component;
  ICompute* get_pB(int portIndex) {
    component->pB[portIndex].providedIntf = component;
    return component;
  }
}
class SystemDelegatee {
  System* component;
  void createConnections() {
    for(int i = 0; i < mul(a); i++) {
      for(int j = 1; j < mul(pA); j++) {
        int partIdx = (i * mul(pA) + j)/mul(pB);
        int portIdx = (i * mul(pA) + j)%mul(pB);
        ICompute* impl = component->b[partIdx].delegatee->get_pB(portIdx);
        component->a[i].delegatee->set_pA(impl);
      }
    }
  }
}
```

- For each $c \in C_{star}$, the connections between the p port and the q port at the other end of the c connector follow the star pattern semantics.

- The array pattern semantics is applied to all connectors in C_{array}.

The hybrid structure allows both of the star and array pattern semantics to be applied to a set of connectors starting from the same port. It means that each invocation of one of the methods required by p is alternatively propagated to the port instances at the other ends of the connectors in C as illustrated in Section 5.3.1 on page 82. In Fig. 5.12 (a) on page 83, the *conn1* and *conn2* connectors start from the same port pA. Requests (method calls) from pA are alternatively propagated to all instances of the pC port (12 ports) and an instance of the pB port, which is specified by the equations 5.5 and 5.6.

To realize this semantics, an intermediate component, namely intermediary, is created to bridge the connections between the p port and the ports at the other ends of the connectors of C. In the delegatee code, p is connected to the intermediary instead of the other end ports. The purpose of the intermediary is to receive requests from the p port and alternatively propagate them to the other ports following the hybrid structure semantics as described above. The intermediary can be considered as a *forwarding point* that receives a request from the p port and alternatively *forward* it to other connected ports.

Fig. 5.14 shows how the intermediary for the example in Fig. 5.12 (a) works. Requests are sent from an instance of pA, $a[1].pA[0]$ in particular, to the intermediary. By using the equations 5.5 and 5.6, the instance of pB that is connected to pA via the *conn1* array pattern connector can be computed. The intermediary is also connected to all instances of the pC port since the *conn2* connector is of the star pattern.

Let's examine the code generation process through the example in Fig. 5.12 (a) on page 83 with its corresponding generated code in Listing 5.6 on page 90 as follows:

1. The first step is to create the $IntermediateICompute$ intermediate class for the required interface $ICompute$ of the port pA. As said, this intermediary maintains a

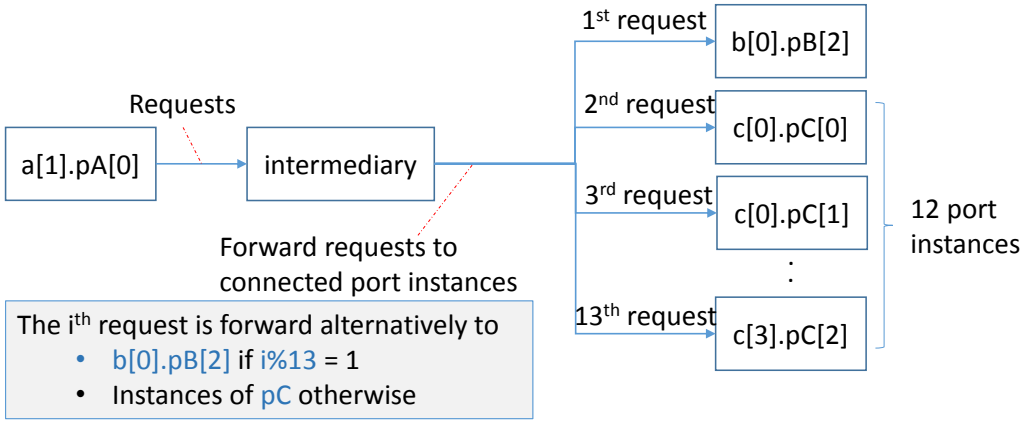


Figure 5.14: Example of using intermediary to bridge the connections between p and b and c

references list of implementations of the required interface (line 10) to which the pA port is connected. The intermediate class implements the $add(int, int)$ method of $ICompute$ to alternatively call the corresponding method of the instances connected to the pA through the *references* attribute.

2. Secondly, a $pA\_A\_intermediate$ attribute typed by the intermediary is declared within the $SystemDelegatee$ delegatee of the container of the *conn1* and *conn2* connectors. The purpose is to connect the required interface attribute of the pA port to this intermediary (lines 3-5) so that requests can be sent from pA to the intermediary.

3. Lastly, the *createConnections* method is created for the system configuration. For each port instance of pA, the method uses the equations 5.5 and 5.6 to compute the instance of pB that should be connected to pA following the array pattern (36-40). For the *conn2* connector, the method follows the star pattern semantics to establish the connections between the intermediary and the pC port instances (lines 41-49). It calls the setters of pA to connect this port to the intermediary (line 49).

By this way, code can be systematically generated. It is worth noting that if all of the connectors in C satisfy the array pattern conditions, the code segment in lines 41-49 for star pattern connections is not generated.

In fact, as earlier discussed, the purpose of the delegatee code is to make the additional structural constructs executable by generating the *createConnections* code from the connector definitions. On the other hand, in an extended code component, only one composite attribute, which is typed by the delegatee class as previously described by the delegation pattern, is added by the T2T transformation. As long as the attribute is not modified by programmers, the extended code complemented by the generated delegatee code is fully compilable, executable, and debug-able by existing IDEs. *"Executable"* and *"debug-able"* here mean that developers can execute and debug, respectively, user-code/fine-grained behavior code where required and provided interfaces of ports are accessed.

We have shown the process of generating code for assembly connectors. The following section discusses how to apply this process to delegation connectors.

Listing 5.6: Generated code for hybrid pattern

```
class ADelegatee {
  A* component;
  IntermediateICompute pA_intermediate[mul(pA)]; //intermediate part
  ICompute* listsOfIntfs[mul(pA)][NUMB_OF_CONNS]; //maintained lists of
    interaces for ports
  void set_pA(ICompute** impls, int portIndex) {
    for(int i = 0; i < NUMB_OF_CONNS; i++) {
      listsOfIntfs[portIndex][i] = impls[i]; //refering implementations
    }
    pA_intermediate.setReferences(listsOfIntfs[portIndex], NUMB_OF_CONNS);
    component->pA[portIndex].requiredIntf = &pA_intermediate;
  }
}
class IntermediateICompute: public ICompute {//intemediate component
private:
  ICompute** references;
  int numberOfRefs; int nextInstance;
public:
  void setReferences(ICompute** refs, int numberOfRefs) {
    this->references = refs;
    this->numberOfRefs = numberOfRefs;  nextInstance = 0;
  }
  void add(int a, int b) {
    references[nextInstance]->add(a,b);
    nextInstance++; //point to the next instance
    if (nextInstance == numberOfRefs) {
      nextInstance = 0;
    }
  }
}
class SystemDelegatee {
  System* component;
  void createConnections() {
    for(int i = 0; i < mul(a); i++) {
      for(int j = 0; j < mul(pA); j++) {//loop over each port
        ICompute* intfs[NUMB_OF_CONNS]; //a list of interfaces
        int partIdx = (i * mul(pA) + j)/mul(pB);
        int portIdx = (i * mul(pA) + j)%mul(pB);
        int count = 0;
        //the first connections to the ports instances of array connectors
        intfs[count++] = component->b[partIdx].delegatee->get_pB(portIdx);
        //the other connections for star connectors
        for(int k = 0; k < NUMB_OF_CONNS - 1; k++) {
          int cIndex = k / mul(pC);
          int pCIndex = k % mul(pC);
          //connect all of port instances of c to listOfIntfs
          intfs[count++] = component->c[cIndex].delegatee->get_pC(pCIndex);
        }
        //call setter for the port
        component->a[i].delegatee->set_pA(intfs, j);
      }
    }
  }
}
```

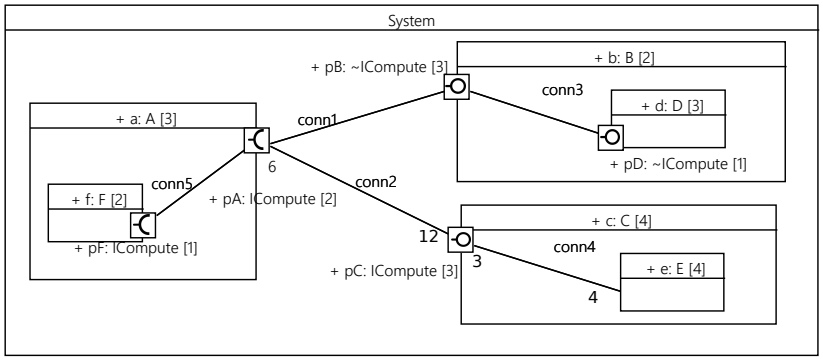


Figure 5.15: Delegation connectors example with star and array pattern

Provided delegation connectors A provided delegation connector transfers the method calls from the provided port of a component to one of the inner parts of the component. For example, the *conn3* connector in Fig. 5.15 delegates from *pB* to *pD* on the *d* part. There can be multiple connectors from the same port of the component to multiple parts with port of the component. Code generation for provided delegation connectors is largely similar to that of assembly connectors by assuming that the component is a virtual part with 1 as its multiplicity and the port of the container acts as a required port. Hence, an intermediary is also created within the component similarly to the case of assembly connectors.

Required delegation connectors A required delegation connector transfers the method calls on the required port of an inner part contained by a component to the component's port. There can be multiple components from the required port to multiple ports of the component. The ports of the component, which might actually connect to other ports, act as provided ports from the point of view of the port of the inner part. Similar to the code generation for assembly connectors, an intermediary is created within the container.

Code generation for flow ports As previously discussed, flow ports enable data flow-based communication schema between components, in which messages modeled as UML Signals are sent and received. We also provide an attribute typed by the pre-defined *IPush<Sig>* interface for flow ports to send out or receive messages. The semantics of the interface is similar to that of a required/provided interface of a service port. Therefore, the in-place text-to-text transformation of flow ports is similar to that of service ports.

5.3.3 Discussion

We have shown a set of code generation patterns used in the text-to-text transformation from additional structural constructs to delegatee code. The novelty is that the patterns make the constructs executable and concretely provide an automated way for producing code conforming to the array and star connection patterns precisely specified by PSCS.

The semantics of the connection patterns is also discussed in [Cuccuru 2008]. Those semantics are later standardized by the OMG in PSCS. In [Radermacher 2009], a model-driven tool chain is proposed to automatically generate code from UML models, that use UML-CS elements. The semantics of ports and connectors are redefined to adapt it to the context of distributed embedded systems. However, no concrete code generation for the

array and star connector patterns is taken into account in [Radermacher 2009]. The support of these connector patterns is also missing in several modeling tools such as Papyrus-RT [Posse 2015], IBM Rhapsody [IBM 2016a] and Enterprise Architect [SparxSysems 2016], although these tools provide code generation from UML ports and parts with multiplicities.

The only work that takes the connector patterns into code generation is in [Ciccozzi 2016a]. This latter supports generation of explicit links among port instances. The array pattern is supported with multiplicities for parts and ports. Furthermore, the author also defines two other patterns, namely *star connector pattern* and *perfect shuffle pattern*. These patterns are actually specialized cases of the array pattern where multiplicities of parts and ports are some special values such as even numbers. Note that, the star connector pattern in [Ciccozzi 2016a] is different from what is understood in this thesis and standardized by the OMG. Leaving out this variation of the star connector pattern, the multiplicities of connector ends are not discussed by the author.

In this next section, we will show the code generation patterns and delegation from state machine constructs.

5.4 Transformation from state machine elements to code

This section describes the text-to-text transformation from state machine programming constructs, e.g. state, transition, event, pseudo state, used in the extended code as previously discussed to delegatee code. This latter complements the extended code so that the proposed constructs are executable. The missions of this transformation are, on the one hand, similar to that of the T2T transformation for the structural constructs: (1) to generate delegatee that is physically separated from each other; (2) to support the two codes being logically related with each other through delegation as described in Section 5.2 on page 78; and (3) to support a complete set of the behavioral constructs. On the other hand, the transformation should generate efficient code for embedded systems. For this latter, we are especially interested in producing code that enables fast event processing and requires a small amount of memory.

The T2T transformation is based on a set of patterns for generating code from state machine elements. Many tools and approaches have been proposed to automatically translate UML-SM elements into executable code in the context of MBSE [Mussbacher 2014]. However, those existing tools and approaches either support a subset of UML-SM elements or handle composite state machines by flattening into simple ones. The latter implies a combinatorial explosion of states, and excessively generated code [Badreddin 2014]. In the context of UML-SM elements, the following specific issues are found in the current UML tools.

- **Completeness:** The existing tools and approaches mainly focus on the sequential aspect while the concurrency of state machines is limitedly supported. Pseudo states are not rigorously supported by existing tools such as Rhapsody [Shinji 2016]. Rhapsody does not support deferred events, *doActivity*, and change-event [Shinji 2016, Harel]. Enterprise Architect [Sparx Systems 2017] supports *doActivity* in a sequential manner. Pseudo states such as history, choice and junction are poorly supported in Enterprise Architect [SparxSysems 2016] and Sinelabore [SinelaboreRT]. Designers are then restricted to a subset of the UML-SM elements during design.

- **Semantics:** The semantics of UML-SM is defined by Precise Semantics for UML

State Machine (PSSM). This standard is not (yet) taken into account for validating the runtime execution semantics of generated code. Rhapsody only allows a *pseudo state junction* to have an outgoing transition [Shinji 2016].

5.4.1 Features

Compared to existing approaches for code generation from UML-SM elements, the T2T transformation here has the following features.

Completeness The transformation supports all state machine elements including all pseudo states and transition kinds such as external, local, and internal. Hence, this transformation also improves flexibility of using UML-SM elements to express the event-driven behavior of components that does not appear in current tools and approaches.

Separation Current tools mix automatically generated code with manually written code (user code), e.g. UML tools place manually written code next to blocks of generated code. Instead, the T2T transformation completely separates the generated code, namely delegatee code, from the user code that appears in the extended code component. In this latter, the user code exists in forms of methods including class methods, state action methods, guard methods, transition effect methods, and change expression methods that are equivalent to UML-SM elements in the bidirectional mapping (see Section 5.1 for more details). When the extended code component needs to process some event, it delegates the event processing to the execution control of the delegatee. This latter, during event processing, if necessary, calls appropriate user code methods, e.g. the delegatee calls a transition effect method during event processing. Again, as previously discussed, this transformation is, to some extent, shares the same idea with the *deep separation* [Zheng 2012]. However, our transformation has a much broader scope than *deep separation*. This latter only supports flat state machines (state machines with only simple states) without pseudo states and transition guards as well as the four event types. *Deep separation* uses the state pattern [Shalyto 2006, Douglass 1999] to generate code while we extend the switch/if pattern [Booch 1998] to avoid dynamic memory allocation and to reduce memory consumption in the state pattern. Furthermore, no synchronization for UML-SM and code is supported in *deep separation*.

Event support The transformation promotes the use of all four UML event types and event deference mechanism, which are able to express synchronous and asynchronous behaviors and exchange data between components.

UML-conformance PSSM defines a test suite with 66 test cases for validating the conformance of runtime execution of code generated from UML-SM elements. We have experimented the transformation-generated code with the test suite. Traced execution results of the test cases comply with the standard and are, therefore, a good hint that the execution is semantically correct.

Efficiency We conducted experiments on two benchmarks to show that code generated from UML-SM elements by the transformation is efficient and can be used to develop resource-constrained embedded software. Specifically, event processing is fast and the size

of executable files compiled from the generated code is small. The experimentation is shown in Subsection 5.6.2.

Concurrency Concurrency aspects in state machines including *doActivity* of states, orthogonal regions, event detection, and event queue management are handled by the execution of multiple threads. The design of the latter is presented in Subsection 5.4.2.

5.4.2 Concurrency

The execution of a state machine of a component instance can consist of multiple actions that run concurrently. Imagine that, during the execution, there is at least one active state that can have an associated *doActivity*. Therefore, there are at least two concurrent actions: (1) an action that listens to events coming to the state machine and processes them; and (2) the action of running the *doActivity* of the active state. By the definitions of UML-SMs, a composite state can contain multiple regions, each of which can also contain other composite states. It turns out that multiple states can be simultaneously active. Therefore, the *doActivity* actions associated with these active states execute concurrently. As a result, the execution of a state machine should consist of multiple threads. Rhapsody does not support *doActivity* actions, thus eliminates the need to have concurrent actions for them. Enterprise Architect [Sparx Systems 2017] supports *doActivity* actions in a sequential manner. The code for a *doActivity* action is not concurrent with entry state actions in Enterprise Architect.

This section describes our design of concurrency aspects of state machines in delegatee code at runtime.

5.4.2.1 Thread-based design

Typically, each state machine instance in Rhapsody runs within a single state machine main thread (often called *super loop*) that consumes events stored in an event queue. The support of Rhapsody for time-events (that are called timeouts in [Harel]) is based on a fixed, predefined framework called Object eXecution Framework (OXF) [IBM Rhapsody 2017].

The concurrency of UML-SMs should be, by native, based on multiple threads including *permanent* and *spontaneous threads*. Each thread runs an action that is concurrent with other actions. Permanent threads are created once and their associated actions keep running even there are no events incoming to the state machine. In contrast, spontaneous threads are spawned and only exist for a while during event processing.

The rationale of this multi-thread design is based the execution semantics of UML-SM. First, we examine which actions should be executed if no events are incoming to a state machine. These actions should run concurrently with the state machine main thread that detects and processes incoming events, thus be associated with permanent threads as shown in Fig. 5.16. Second, there might have multiple actions executed during the processing of an event, we investigate which actions should occasionally run concurrently with each other, e.g. when entering a concurrent state with two orthogonal regions, the state entry actions of the active sub-states of the two regions should be concurrently executed. Following the run-to-completion semantics of UML-SM [Specification 2015], these actions should complete before the state machine can process other events in the queue, thus be associated with spontaneous threads. The use of spontaneous threads for actions conforms to the UML specification [Specification 2015] since these actions must conceptually run concurrently.

In most of existing tools such as IBM Rhapsody and Enterprise Architect, the state entry actions of the sub-states of orthogonal regions are sequentially executed.

According to the UML specification, only *doActivity* actions of active states of the state machine are executed. Other *doActivity* actions should be suspended until their states become active. It turns out that there is a need to have a mechanism that suspends or activates the permanent threads. This mechanism is based on a paradigm **wait-execute-wait**. In the latter, a permanent thread **waits** for a signal to **execute** its associated method/action and goes back to the **wait** point if it receives a stop signal or the execution of its associated thread method completes. We identify the actions associated with permanent threads as follows:

- *doActivity*: As previously described, each *doActivity* is associated with a permanent thread.
- *Time-event detection*: The detection of time-event occurrences needs a timer to counts its associated wait-period. This timer should run even if no event is incoming to the state machine. Therefore, a permanent thread is created for a time-event. This permanent thread has a method that counts ticks and emits the event after the wait-period associated with the time-event expires (see Subsection 5.4.3 on page 96 for more details).
- *Change-event detection*: Each change-event is associated with a function, namely *change-event detect function*, for monitoring and evaluating the previous and current values of the boolean expression of the change-event to decide whether an emission of the change-event should be launched. The change-event detect function observes a variable or a boolean expression and pushes an event to the queue if there is a value change of the expression from *false* to *true*. Since the observation does not depend on whether the state machine has incoming events, a permanent thread is assigned to a change-event.
- *Super loop*: The state machine main thread is associated with the state machine method that is commonly called the *super loop* method. It reads events from the event queue and calls appropriate event handlers. Furthermore, this main thread also controls other permanent threads based on the **wait-execute-wait** paradigm. In particular, the main thread sends **start** and **stop** signals to permanent threads associated with *doActivity*s and time-events.

Spontaneous threads are used for the following cases:

- *Effects of transitions*: During event processing, multiple transitions outgoing from a *pseudo state fork* to vertexes of the orthogonal regions of a concurrent state can be activated. The UML specification states that the effects of these outgoing transitions are "*executed concurrently with each other*" [Specification 2015]. Therefore, a spontaneous thread is assigned to each effect. Similarly, the effects of transitions incoming to a *pseudo state join* are concurrently executed, thus each of which is associated with a spontaneous thread. In most of existing approaches [IBM 2016a, SparxSysems 2016, Spinke 2013], these effects are executed in a sequential manner, thus not faithful to the UML specification.
- *Entry actions of sub-states*: When entering a concurrent state, after the execution of its entry action, the entry actions of the active sub-states of its orthogonal regions

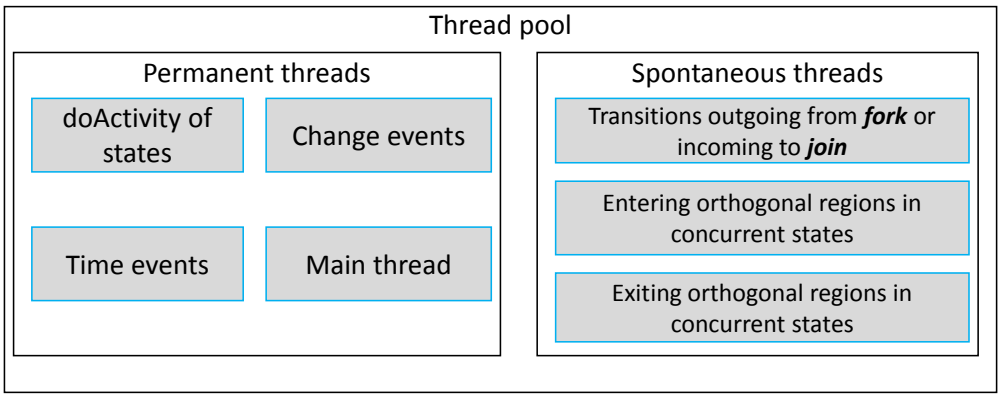


Figure 5.16: Thread-based concurrency design for state machine execution

should be concurrently executed. Therefore, a spontaneous thread is created for each orthogonal region to manage the entering of the region running concurrently with that of the other orthogonal regions.

- *Exit actions of sub-states*: Similar to entering, when exiting a concurrent state, before the exit action of the state, the exit actions of the active sub-states of the concurrent state are executed concurrently. A spontaneous thread is therefore created for each orthogonal region to exit the corresponding region.

5.4.2.2 Thread communication

Each permanent thread is associated with a mutex for synchronization in the multi-thread-based generated delegatee code. The mutex must be locked before the method associated with the thread is executed.

Run-to-completion The event processing must follow the run-to-completion semantics of UML-CS. It says that the state machine completes processing of each event before starting the processing of the next event. If all events are asynchronous, the main thread processes events by reading one-by-one from the event queue. However, because we allow call-events to be synchronous, which means that the processing of call-events happens in the threads of components that call methods associated with the call-events. Therefore, the processing of synchronous and asynchronous events can violate the run-to-completion semantics. To avoid it, a mutex, namely *main mutex*, is associated with the main thread to protect the run-to-completion semantics. Each event processing must lock the main mutex before executing the actual event processing. This is to ensure that at any moment, there is at most one event being processed regardless of in which thread the processing happens. In generated delegatee code, lock and unlock are implemented using the convenience of signals and conditions in POSIX [Butenhof 1997].

5.4.3 Code generation pattern-based separation of state machine extended code and delegatee code

This section describes our code generation pattern for states, regions, events, and transitions.

5.4.3.1 Separation for state action

Commonly, a UML state is transformed into an enumeration literal element in SWITCH/IF/ELSE-based approaches [Booch 1998] or a class in state pattern-based approaches [Niaz 2004, Spinke 2013]. The latter can generate code from hierarchical state machines but entail the use of dynamic memory allocation, which is not preferred by embedded systems. Furthermore, the authors in [Charfi 2012] show that state pattern-based generated code consumes much memory. In this thesis, a common state type *IState* is created. The type has two attributes, namely *actives* for active sub-states of the hierarchy of composite states and *previousActives* for referring to previous active sub-states in case of the presence of history states. For each UML state, an *IState* instance is created and a state identifier is assigned to each state. During initialization, each instance initializes its attributes to a default value meaning inactive state.

Listing 5.7 shows the state type and its instances where *STATE_MAX* indicates the number of states and is computed for each state machine[1]. Three methods, namely *entry/exit/doActivity*, are created for calling state action methods in the extended code by using the *component* reference as shown at lines 9-16 of Listing 5.7. An example of the interaction between the component in the extended code and its delegatee is shown in Fig. 5.18.

Listing 5.7: IState type in C++

```
1  typedef struct IState {
     int previousActives[2];  int actives[2];
3  } IState;
   class FIFODelegatee {
5  private:
     FIFO* component;
7    IState states[STATE_MAX];
   public:
9    void entry(StateId id) {
       switch{id} {
11     case SIGNALCHECKING_ID:
         component->entryCheck();
13       break;
         //code for other states
15     }
     }
17 }
```

As previously discussed in Subsection 5.4.2, multiple *doActivity* actions of states might concurrently run during the execution of the state machine if this latter has composite states. For the state machine example in Fig. 5.1 on page 67, the *doActivity*s of the *DataQueuing* and *Queuing* state can be semantically run simultaneously. Each *doActivity* is then run within a permanent thread and a mutex is created for controlling it as designed in Subsection 5.4.2. Listing D.1 on page 135 shows a code segment example for *doActivity* threads.

5.4.3.2 Region

A region contains multiple vertexes and transitions. Semantically, during state machine execution, a region of a composite state can be entered in several ways:

- **Entering by default**: A transition ends at the border of a composite state. By this way, this latter is first entered and its entry action is executed. The initial state of the

[1]To avoid runtime memory allocation, STATE_MAX is computed for each state machine, rather than for all state machines, which will waste memory for small state machines.

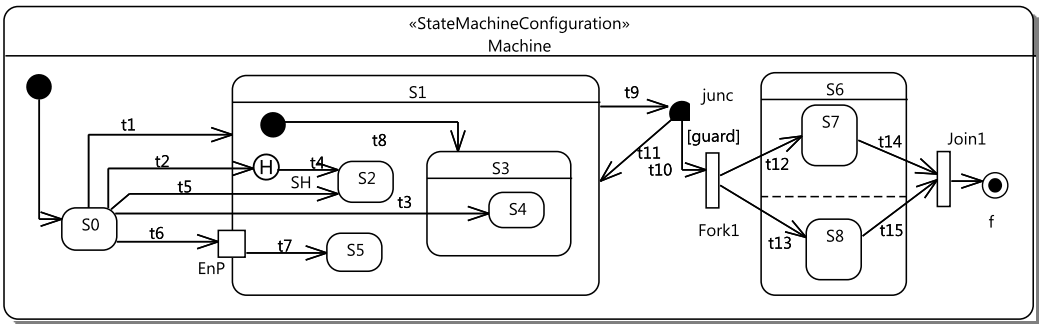


Figure 5.17: Example illustrating different ways entering a composite state

region is then entered. For the example in Fig. 5.17, the region of the *S1* composite is entered by default through the *t1* transition.

- **Border-crossing transition**: A transition ends at a direct or an indirect sub-vertex of the region. As in UML, the entering of the region follows the activation of the composite state and the sub-vertex then becomes active. The transitions *t2* and *t5* in Fig. 5.17 illustrate such border-crossing transitions.

Papyrus-RT [Posse 2015] generates code from state machines using IF/ELSE statements but it does not support orthogonal regions as well as border-crossing transitions. Papyrus-RT requires that a transition to a composite state ends at an *entry point* of the composite state. IBM Rhapsody [IBM 2016a] and Enterprise Architect [SparxSysems 2016] have support for orthogonal regions and region entering ways but it does not have a separation of the extended code and delegatee code as in this thesis.

We consider regions as first elements each of which is transformed an entering and an exiting method for controlling the ways the region is entered and exited. The details of how the code for the methods is generated are shown in Listing D.2 for keeping the main content of thesis focused at the right abstraction level.

5.4.3.3 Event

An event in UML is a notable occurence at a particular point in time [Specification 2015]. Niaz et al. [Niaz 2004] supports the mapping of time-events and signal-events to Java. They extend the state pattern [Shalyto 2006] to represent a state as a Java class. For time-events, Niaz et al. requires that the class associated with a state relevant to a time-event implements a predefined interface *TimedState*. This latter has a method *timeout* that is invoked as soon as the wait-period of the time-event expires. Niaz et al. assume that there is a single-event queue for the whole system. A signal-event is sent from a sender to a receiver by placing the event in the queue. The system then dispatches the event to the right receiver for processing [Niaz 2004]. Rhapsody relies on the Object Execution Framework to detect time-event occurrences and a *gen* operation to generate and manage signal-event occurrences.

Differently from the approach in [Niaz 2004], we have an event queue for each component instance. In the delegatee code, we provide the design for the detection of time-events and change-events based on permanent threads as described in Subsection 5.4.2 on page 94. Furthermore, we provide an implementation for detecting change-events that are not

supported in the existing approaches. Specifically, the following list briefly describes how the detection of the four event types is implemented in the delegatee code.

- **Call-event**: When its associated operation is called, an instance of the call-event is created. Typically, a call of the operation associated with the call-event is invoked through a service port of the extended code component containing the operation. As previously described, the processing of call-events can be either synchronous or asynchronous depending on the design. For a synchronous call-event, the event processing method waits and locks the main mutex protecting the run-to-completion semantics as previously mentioned, and executes the event processing code (see 5.4.2). The *synchronous event processing* is realized by *directly calling the event processing code* since the processing must be run within the thread of the operation caller. The details of how the event processing code is generated are described in Section D.3 in Appendix D on page 135. Listing 5.8 on page 100 shows the event processing method generated from the *DataPushEvent* call-event in the service port-based producer-consumer example. Note that the only way to call this processing event method is to invoke the *push* method associated with this call-event through a service port as shown in Subsection 5.3.2.

  Rhapsody also provides synchronous call-events in term of *triggered operations* [Harel]. In the latter, the authors also note the deadlock problem of synchronous call-events. It occurs when there is an invocation of the operation associated with a call-event in the midst of performing a transition effect [Harel]. Therefore, developers should pay attention to the use of synchronous call-events to avoid the deadlock problem. Asynchronous call-events do not have this problem because event instances are put to the event queue and the event processing happens in the thread of the event receiving state machine.

- **Signal-event**: An API *push* is created for other components to send an instance of the signal associated with the event by calling it through a data-flow port. As previously described in Section 5.1 on page 66, for the flow port-based producer-consumer example in Fig. 5.4 on page 72, when the *push* method is called via the *pOutData* port of the *p* part, the *push* method of *FIFO* is called, a signal-event is then emitted and written into the event queue managed by *FIFO*.

- **Time-event**: As previously discussed in Subsection 5.4.2, a thread associated with the time-event is created and initialized at the initialization. The thread then enters a **wait-for-start** point. Assume that a state S has at least an outgoing transition triggered by the time-event. Within the thread execution, the method associated with the thread waits for a *start* signal, which is sent after the execution of the entry action of the S state completes, to go to a **sleeping** point. At this point, the thread sleeps for a wait period specified by the time-event. When the wait period expires, an event instance is emitted and written to the event queue if the state is still active. The thread then goes back to the **wait-for-start** point. If it receives a *stop* signal before the wait-period expires (i.e. while sleeping), it goes back to the **wait-for-start** point without emitting any event.

- **Change-event**: Similarly to time-events, a thread is initialized and its method waits for a re-evaluation signal. As previously mentioned, the boolean expression of a UML change-event is transformed into a change method in the extended code component.

Note that the method of the thread assigned to the change-event is of the delegatee class. This change-event thread method checks whether the value of the boolean expression of the event is updated from false to true by calling the change method through the *component* reference. If so, an event instance is sent to the event queue.

Listing 5.8: Event processing method for the *DataPushEvent* call-event in the service port-based producer-consumer example

```
class FIFODelegatee: public IPush {
public:
  //..
  void processDataPushEvent(Data& data) {
    if (active_state == IDLE_ID) { //check the idle state active
      component->signalCheck(data); //call the transition effect method
      active_state = SIGNALCHECKING_ID; //change the active state
    }
  }
  //..
}
```

As above presented, all asynchronous incoming events are stored in a runtime priority queue, in which each event type has a priority. The completion-event always has the highest priority. Others are equal by default.

Note that the code generated for the event types is put within the delegatee associated with a component. The generated code uses information related to events such as wait period of time-events. Since the generated delegatee code is hidden from programmers perspectives, the only way to change it is to modify the state machine code in the extend code and then re-execute the T2T transformation. This latter is therefore a powerful means that makes the additional constructs being considered executable from the perspectives of programmers.

5.4.3.4 Transitions and pseudo states

A transition in UML can go from a vertex to any other vertex in the state machine regardless of the depth of the state machine hierarchy. Transitions can be combined with pseudo states and events to make them powerful for modeling dynamic aspects of reactive systems. However, most of the existing approaches do not implement all pseudo states as noted on page 92. Furthermore, none of the existing approaches supports the synchronization for transitions and pseudo states. A transition can be implemented as SWITCH/IF/ELSE statements, a combination of source state and event in a state table pattern [Douglass 1999], or even a method in an object-oriented pattern [Spinke 2013]. This latter, however, uses dynamic memory allocations. The state table pattern only deals with flat state machine. However, when there are multiple transitions that have the same source vertex, have different guards and are triggered by the same event, the state table pattern fails to deal. It is because this pattern only uses the source vertex and the event to specify a transition. We extend a SWITCH/IF/ELSE-based pattern to provide code generation with full support for transitions.

The challenges in dealing with code generation for transitions and pseudo states while taking the synchronization into consideration are identified as follows:

- Full support for transitions and pseudo states and semantics conformance: An implementation pattern for transitions and pseudo states is proposed. This pattern takes into account the semantics of UML-SM according to PSSM.

- Complete separation of code generated for transitions and pseudo states within the delegatee code from the extended code: The delegation from the extended code state machine to the delegatee code is used.

The implementation pattern for transitions and pseudo states involves many implementation-related details. In order to keep the main part of the thesis at the right abstraction level, such details are put into Section D.3 on page 136 of Appendix D on page 135.

Signal-event processing example Fig. 5.18 on page 102 shows the interactions between the *fifo* part and its delegatee during processing of an incoming signal-event. Here, we use the flow port-based producer-consumer example with signal-events as in Listing 5.1 on page 72 for illustration. The discussion of the service port-based one is very similar. Let's assume that an instance of the *Data* signal is sent from the producer to the fifo through the *pInData* port of *FIFO* (see Listing 5.1 on page 72) and that the current active state of the *FIFOMachine* state machine is *Idle*. Upon reception of the signal instance, a signal-event is emitted and saved to the event queue managed by the fifo *delegatee* instance. This latter reads the event from the queue and starts the processing. The latter checks whether the current active state is *Idle* and then invokes the *signalCheck* transition effect method implemented in the fifo via the *component* reference. The event processing then changes the current active state to *SignalChecking* and uses the *component* reference to call the *entryCheck* entry action method implemented in the fifo. Upon completion of the event processing, the *fifo delegatee* goes back to a wait-point for reading next events from the queue.

Support for non-deterministic transitions It is possible multiple transitions from the same source vertex (state or pseudo state) are triggered by the same event. In this case, there is ambiguity for selecting which one of the enabled transitions should be activated. Consider the example in Fig. 5.17 on page 98 and assume that the transitions *t1*, *t2*, and *t5* with *guard1*, *guard2*, and *guard5* as their respective expression guard, can be triggered by the same event e_{S0}. If *S0* is active and an instance of e_{S0} is received by the state machine, non-determinism occurs if at least two of the three guard expressions become true. We assume the values of *guard1* and *guard5* are true at runtime in this case. Either *t1* or *t5* should be activated and the next active configuration of the state machine is different from each activation. *S3* is active if *t1* is activated and *S2* is active otherwise. For transition selection in this situation, we propose three options to deal with the non-deterministic transition selection, as follows:

1. Priority by creation: Among the enabled transitions, the transition created (by modelers) first is chosen to be activated. If *t1* is created before *t5*, it is selected for activation.

2. Random selection: One of the enabled transitions is randomly selected to be activated.

3. User configuration: A UML profile is created and allows users/modelers to explicitly specify which transition has higher priority. The transition has the highest priority in the enabled set is selected.

The Moka model execution engine [Papyrus 2016] also intends to support several options for transition selection in case of non-determinism. The current implementation of

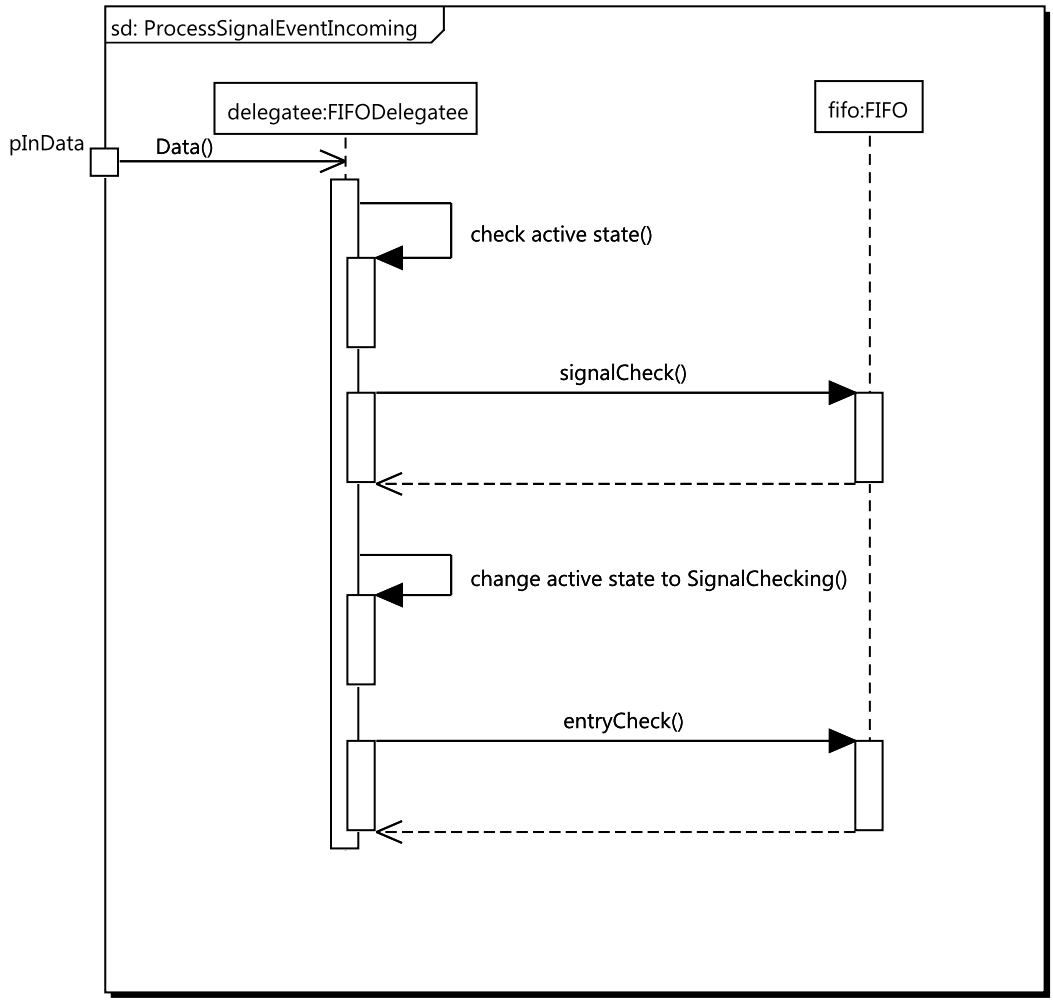


Figure 5.18: Interactions between *FIFO* and its delegatee during processing of a signal-event

Moka supports the first option [Papyrus 2016]. We use Moka for our evaluation for UML-conformance of generated code in Subsection 5.6.1 on page 108 since it provides a precise execution semantics for both UML-CS and UML-SM elements.

5.4.4 Discussion

The introduction of additional constructs into an existing programming language provides it with more functionalities while keeping source code written in the programming language executable. It is investigated in a few approaches. In [Hinkel 2015], the authors discuss the trade-off between the use of general-purpose programming languages (GPL) for implementing model transformations vs the use of specialized model transformation languages. They argue that, on the one hand, GPLs like Java are widely used by developers but not suitable for model transformations. On the other hand, model transformation languages like QVT provide high-level concepts suitably for efficiently expressing the intent of developers but are not popular in industry. They then propose an *internal model transformation language* within an existing GPL, C# in particular. The existing GPL is called *hosting language*. The same argument is applied to our approach: the standard programming language is hardly applied to manage system complexity that can be better addressed by the additional programming constructs. The latter can be considered as an *internal language* while the standard programming language as a *hosting language*.

This principle is also applied to the recently proposed *P* [Desai 2013] and *P#* [Deligiannis 2015] programming languages. *P* and *P#* embed a small subset of UML-SM elements, in particular the state and transition concepts, into C and C#, respectively, for dealing with complexity. *P* and *P#*, however, do not intend to conform to UML as our approach does. In [Henry 2008], UML-SM elements are embedded into C++, namely *meta state machine*. We compare this latter with our approach by seeing from them from different perspectives. Seeing them from a perspective of supported UML-SM elements, the extended language has a broader scope. As previously presented, we provide a complete set of UML-SM elements represented in C++. Meta state machine, however, does not support pseudo state *join, choice, junction*, signal-event, time-event, and change-event. Speaking about state machine-based design, we use a UML-SM to define the behavior of a component that can interact with other components through its ports. On the other hand, each state machine in the meta state machine is not associated with any component. Regarding the implementation, state machine elements in the extended code are used for generating *dynamic library code*, namely delegatee code as previously described, by using a code generation pattern that is customizable (another code generation pattern can be used instead). Here, *dynamic* means to only generate the delegatee code for the used state machine elements. In contrast, the meta state machine encodes the supported state machine elements into a static library. In fact, it uses an internally "hard-coded" code generation pattern that is hard to replace. Besides, the syntax of the state machine in the extended code is flexible and can be adapted to developers' preference. Subsection 5.1.2 on page 73 shows our proof-of-concept for the syntax that is similar to that of some textual modeling languages such as Umple. In contrast, the approach in [Henry 2008] proposes a rather complicated syntax.

We argue that, our approach has broader support than these approaches in the sense of completeness of the UML-SM elements. Furthermore, our goal is to use the additional constructs, which correspond to elements of UML-SM, for synchronizing code with UML-based architecture model in MBSE rather than their code-centric approaches.

5.5 Application of the synchronization mechanism by providing use-cases

As described in Chapter 4, a generic methodological pattern for the synchronization of concurrent modifications of model and code is proposed. This synchronization mechanism especially requires the availability of the following use-cases:

- **Batch code generation**: generates and overwrites any existing code from model.
- **Incremental code generation**: updates the code by propagating changes from the model to the code.
- **Batch reverse engineering**: creates and overwrites any existing model from code.
- **Incremental reverse engineering**: updates the model by propagating changes from the code to the model.

The definitions of these use-case are found in Subsection 4.1.2. The batch code generation and reverse engineering are straightforwardly supported by using the proposed bidirectional mapping between the architecture model and code. The incremental code generation and incremental reverse engineering need a classification and management of modifications made in the model and code.

Incremental code generation Table 5.3 shows our actions for propagating model modifications to code. We distinguish structural and behavioral modifications (since the structural and behavioral elements correspond to different code parts), which result in creating/removing/regenerating the corresponding code part. Although only add/remove/update modifications are detected, the moving of a model element can be detected as a combination of a removal of the element from an old container, followed by an addition of the element to a new container.

In fact, modifications related to Remove/Update a type (Class/Component/Inteface) might require regeneration of multiple other types that are dependent on the modified type. However, the re-transformation is not needed in some cases because the delegatee code associated with an extended code component only depends on the additional constructs used in the extended code component. Therefore, if the modified type does not have any effect on the additional constructs used within the dependent components, e.g. the required/provided interface of a service port or the signal of a flow port, the re-generation of the delegatee code for the dependent components is not needed. For example, a class A is renamed and a class B has an attribute typed by A. The renaming of A requires the regeneration of class B while the regeneration of the delegatee code for B is not needed. If there is a deletion of a type, a class containing attributes typed by the deleted type should be regenerated to avoid unknown type problems during compilation.

Incremental reverse engineering Our incremental reverse engineering is similar to change translation [Hettel 2010] and change-driven transformation [Ráth 2009]. The latter listens to changes made in a model and uses predefined rules to propagate the changes back to another model. However, change-driven transformation cannot be applied directly to propagate changes in code back to the model because the detection of changes in code is non-trivial. The approach in [Kramer 2015b] records every developer operation by creating

Table 5.3: Model change classification and management

Element type		Modification	Action
Structure	Part/Port/ Connector	Add/ Remove/ Update	Regenerate the extended code and re-transform it to update the delegatee code for the extended code component containing the modified model element.
	Class /Component /Interface	Add/ Remove/ Update	Create/Remove/Update the corresponding code file(s). If the modification is remove or rename (update), regenerate the extended code of the classes/components/interfaces that depend on the removed/renamed element.
	Property	Add/ Remove/ Update	Regenerate the corresponding extended code class
Behavior	Operation	Add/ Remove/ Update	Regenerate the corresponding extended code class
	UML state machine	Add/ Remove/ Update	Regenerate the extended code and re-transform it to update the delegatee code for the extended code component that contains the modified state machine

a dedicated code editor. However, this approach is not reliable because it is hard to monitor all of developer modifications if the developer uses different editors to modify the code.

In our approach, we do not record all of modifications made to code elements. In contrast, we use a *File Tracker* to detect which files are changed by developers. This kind of tracking is much easier to realize and more realistic than the above approaches. More importantly, it is also supported by several tools such as Git (that supports **tracking file changes**, among other features). The details of our approach are shown in Fig. 5.19.

The file tracker monitors all of the extended code files generated from the architecture model. In fact, the tracker does not listen to changes made to fine-grained programming language elements such as renaming an attribute, adding a method or changing a statement. After all modifications have been made in the extended code, the tracker returns a list of files which have been changed. We do not allow renaming or deleting a class because doing these modifications at the code level requires doing some additional re-factorings. For example, deleting a class requires re-typing class attributes typed by this deleted class. We believe that working at the model level is more suitable for these modifications because the re-factorings can be done through code re-generation from the modified model. The incremental reverse engineering then propagates code modifications within the class scope (ports, attributes, methods, state machines) back to the model. It means that renaming and deletion of elements *inside* a class are supported but renaming of a class is not.

The modified files and the model are then used as input for reverse engineering to update the model. For each modified file, the incremental reverse engineering for each code element in the file follows a **Update-Create-Delete** strategy described in Algorithm 2. The latter is explained as follows:

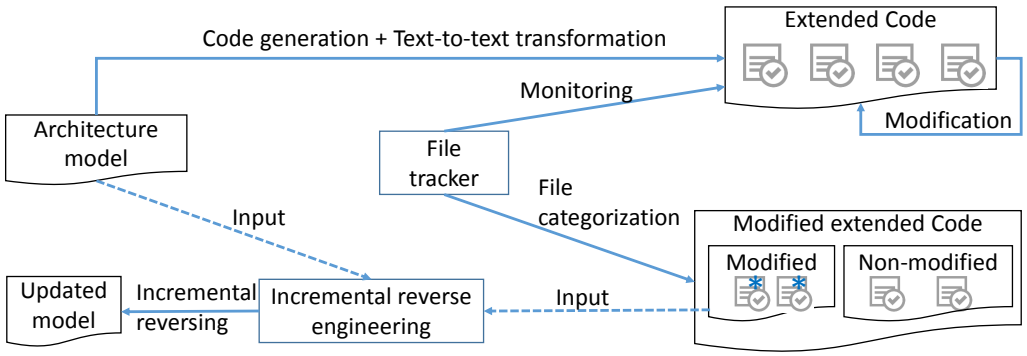


Figure 5.19: Incremental reverse engineering with file tracker

- **Step 1 - Update-Create classes in model** (lines 4-6): Find a class *umlClz* in model matching the class *clz* in code by using its name and hierarchy. If *umlClz* is found, use the information of the code element to update the model element. Otherwise, it means that the class *clz* has been added to code since class renaming and deleting are not allowed. Therefore, a class is created in the model (line 6).

- **Step 2 - Update-Create model elements within class**: A similar strategy is applied as the first step. For each code element (attribute/method/port/state machine element/binding/event) of a class in code (line 8), the information of the code element including name, element type and related information (signature for method) is used for updating or creating a corresponding model element (lines 9-11). For example, if we modify the method body of *entryCheck* in Fig. 5.5 on page 73, the incremental reverse engineering will automatically propagate the changed body to the architecture model as a block of text. If a programmer adds a state to the state machine example in Fig. 5.5, a UML state will be created in the model. The updated or created model element is then added to a list of elements, namely *touchedModelEles*. This latter contains model elements that have associated code elements.

- **Step 3 - Delete**: The *touchedModelEles* list contains UML elements (attributes, ports, connectors, methods, state machines, bindings, and events), which have associated elements in the code. Therefore, if a model element is not found in this list, it should be deleted (lines 13-15).

It is worth noting that the incremental reverse engineering detects a renaming in code as an addition followed by a deletion at the model level.

In the next section, we describe our experiments to evaluate the approach.

5.6 Evaluation results

Implementation Section 4.3 on page 56 in Chapter 4 describes an Eclipse-based instantiation of the proposed model-code synchronization methodological pattern for UML 2.5 and C. The implementation reuses the existing Eclipse-based technologies, namely, Eclipse CDT, EMF-IncQuery and notably Papyrus [Gérard 2010] and the Papyrus component software designer [LISE]. The resulting UML-C++ synchronization tool is integrated into the latter. In the sequel, we describe the building blocks already available and the differences

Algorithm 2 Propagation of code modifications in a modified file to model

Require: A modified file containing a set of object-oriented classes $ModC$ and a UML model M to be updated

Ensure: Modifications within $ModC$ are propagated back to model M

```
1: procedure PROPAGATEMODIFICATIONS(ModC, M)
2:     CodeClasses ← An empty list of code classes;
3:     for clz ∈ ModC do
4:         umlClz ← findModelClass(clz);                          ▷ Update a class
5:         if umlClz == NULL then
6:             umlClz ← createModelClass(clz);                    ▷ Create a class
7:         touchedModelEles ← An empty list of UML elements;
8:         for codeEle ∈ getCodeElements(clz) do
9:             umlEle ∈ findModelElement(codeEle, umlClz);        ▷ Update model element
10:            if umlEle == NULL then
11:                umlEle ← createModelElement(codeEle, umlClz);  ▷ Create model element
12:            touchedModelEles ← touchedModelEles ∪ {umlEle};
13:        for umlEle ∈ getModelElements(umlClz) do
14:            if umlEle ∉ touchedModelEles then
15:                Delete umlEle;                                 ▷ Delete model element
```

compared to our work. Formerly, Papyrus software designer also allows using a subset of UML-CS and UML-SM elements for component-based design and features full C++ code generation through embedding of fine-grained code as blocks of texts. It allows to use some time notions from the MARTE [OMG 2011] profile to specify the wait-period of time-events. Papyrus software designer did not support a model-code synchronization as in this chapter. The approach presented in this chapter for model-code synchronization of component-based reactive system design and implementation is integrated into Papyrus software designer and implemented on top the previous UML-C++ synchronization extension. C++ generated code runs within POSIX systems such as Ubuntu, in which *pthreads* are used for implementing threads for concurrency of UML-SM. This section presents the evaluations of results of the approach through experimentation.

Evaluation The experimentation-based evaluations presented in this section are related to the requirements described in Section 2.2 on page 13. For the UML-based reactive system design requirements in Table 2.2 on page 15, the R6 and R7 requirements related to the completeness of supported UML-CS and UML-SM elements are satisfied since the T2T transformation patterns support a complete set of the UML-CS and UML-SM elements. Therefore, we present here the following evaluations:

- **UML-conformance**: This evaluation is to assess the R8 requirement. We assess the conformance of the runtime execution of generated code to the UML semantics.
- **Efficiency of generated code**: This evaluation is to assess the R9 requirement. We assess the performance and memory consumption of generated code since these aspects are important in embedded systems.
- **Feasibility of the approach**: We assess the feasibility of the application of the

model-code synchronization methodological pattern presented in Chapter 4 and the bidirectional mapping between model and code proposed in this chapter in a case study.

5.6.1 Semantic conformance of runtime execution

This section presents our results found during experiments with Papyrus to answer the following research question.

> ***Research question 1:*** *Is the runtime execution of code generated from UML-CSs and UML-SMs by Papyrus semantic-conformant to PSSM?*

To evaluate the semantic conformance of runtime execution of the generated code, we do unit testing by using the test suites provided by PSCS and PSSM. PSCS and PSSM are standardized by the OMG to provide a precise execution semantics for UML-CS and UML-SM, respectively. Each of them provides a test suite that contains many test cases that a model execution and simulation engine should pass to demonstrate its conformance to the standards. These standards are defined to avoid different semantics variations [Shinji 2016, Fecher 2005, Luo 2016, Deligiannis 2015, Desai 2013, Knapp 2004].

Moka [Papyrus 2016] is a model execution engine offering PSSM and PSCS. As Papyrus software designer, it is an additional component for the Papyrus modeler. Fig. 5.20 shows the evaluation method. The latter consists of the following steps:

Step 1 For a **Test case** from the PSCS and PSSM test suites, we simulate its execution by using Moka to extract a sequence **Trace 1** of observed traces including executed actions.

Step 2 From the same test case, we generate the C++ **Extended code** for execution. Note that the code generation process here includes generating the extended code and executing the T2T transformation on this generated extended code.

Step 3 The sequence (**Trace 2**) of the actual runtime execution traces is obtained by compiling and executing of the code generated in Step 1.

Step 4 *Trace 1* and *Trace 2* are compared. The semantic-conformance test case is asserted if **Trace 1** and **Trace 2** are the same [Blech 2005].

Note that, the execution trace is obtained by simply logging execution steps, e.g. the execution of a state action, as strings to the standard output stream in C++, e.g. *std::out*. Thus, the comparison of the execution traces is only string-based comparison and thus much simpler than conformance checking techniques or model checking techniques. However, to the best of our knowledge, the application of these techniques to the problem of *verifying a C++ implementation conforming to the UML-CS and UML-SM semantics with full concepts* is far from obvious and still a challenge. It is also complex even for the conformance between Java and a very simplified state machine [Blech 2005] even though Java is much better supported by conformance checking tools than C++ [Zhang 2014, Rahim 2010]. This difficulty of the verification mainly originates from: (1) the abstraction gap between the model elements and the code elements; (2) the complexity of generated code verification [Rahim 2010]; (3) the concurrency aspect of UML-SM; and (4) the complexity of the C++

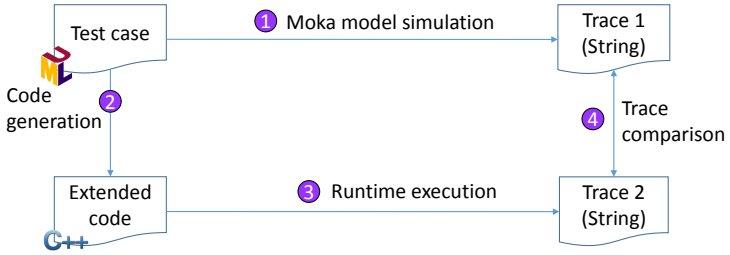


Figure 5.20: Semantic conformance evaluation methodology

Table 5.4: Sub-test suites and the number of test cases of the PSCS test suite

Sub-test suite	Instantiation	Communication 1	Communication 2	Destruction
Test cases	8	14	8	3

programming language itself that makes tooling support for it hard. We expect this situation will be improved in future. Therefore, the evaluation method is chosen because of the availability of tooling support and the ease to realize, rather than the rigor of the testing results. Furthermore, as noted in the PSCS specification [OMG 2015], Moka and the test suites allow us to demonstrate the compliance of generated code in Papyrus with the specifications if no assertion fails when executing the test cases.

PSCS-conformance evaluation Table 5.4 shows the structure of the PSCS test suite. It is divided into sub-test suites for validating certain aspects of the execution of UML-CS. The sub-test suites are described as follows:

- **Instantiation** focuses on the instantiation semantics of composite structures and the management of multiplicities of elements.
- **Communication 1** focuses on the propagation of requests across ports through connectors, especially the request propagation through delegation connectors.
- **Communication 2** is similar to **Communication 1** in the sense of focusing on communication aspects of the UML-CS elements. Furthermore, it specifically consists of test cases for testing the reactions when some request is received on a specific port.
- **Destruction**: focuses on the destruction semantics of UML-CS.

Formerly, the instantiation and destruction sub-test suites have been partially available in Papyrus software designer. We extended this latter to support multiplicities of elements. Since this thesis focuses on the model-code synchronization and partially on dynamic execution aspects of the model and its generated code, we only assert the runtime execution of generated code through the **Communication 1** and **Communication 2** sub-test suites.

To illustrate the evaluation method, let's go into details of an example of the **Communication 1** sub-test suite shown in Fig. 5.21. The *IImpl* interface has a *setP(Integer)* operation that is implemented by the B component. The implementation of the method *setP* within the class B assigns the passed parameter value to its attribute x. When calling *setP* with *4* as input parameter on the p port of C, the call should be forwarded by a forwarding chain. In the latter, the invocation is alternatively propagated to the instances of

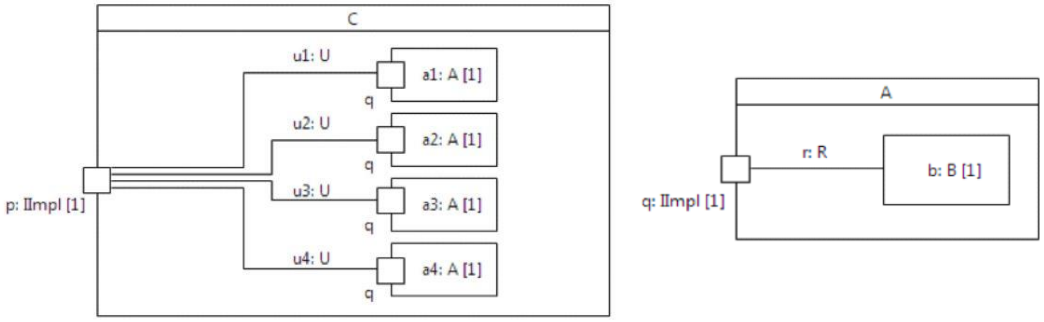


Figure 5.21: An example of multiple delegation with multiple port-port connectors of PSCS excerpted from [OMG 2015]

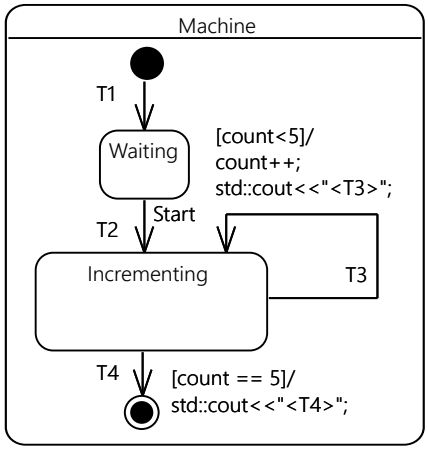


Figure 5.22: An illustrative example of test cases for PSSM

A and then delegated to the *b* instance in *A*. In short, the test case is asserted if: `c.a1.b.x = 4 || c.a2.b.x = 4 || c.a3.b.x = 4 || c.a4.b.x = 4`. This is **Trace 1** resulting from the simulation in the step 1. Therefore, for this test, the execution of the generated code is considered as "passed" if *Trace 2* is the same as *Trace 1*.

In applying the evaluation method to the **Communication 1** and **Communication 2** sub-test suites including 22 test cases, we found that the runtime execution of the generated code for each test case produces the same result as that is monitored during the simulation of the test case.

PSSM-conformance evaluation The PSSM test suite consists of 66 test cases for different state machine element types. The test cases cover the following cases: choice (3), deferred events (6), entering (5), exiting (5), entry(5), exit (3), event (9), final state (1), fork (2), join (2), transition (14), terminate (3), others (2), behavior (6). All of the test cases of the PSSM test suite pass.

Fig. 5.22 shows an illustration model example for the evaluation. The simple state machine describes the process of incrementing a counter *count* from 0 to 5. The increment starts if the state machine receives a *Start* signal-event. this event is emitted if the component of the state machine receives a *StartSignal* signal. Listing 5.9 shows the generated

extended code containing the state machine that is written in the additional behavioral constructs and that also contains the delegetee part. The test case is asserted if the generated code runtime execution trace is *<T3><T3><T3><T3><T3><T4>* since the effect of the *T3* transition is repeated 5 times.

The results of this UML-conformance evaluation of code generated from UML-CS and UML-SM elements are not enough to prove that the implementation pattern and the tooling support preserve the execution properties of UML-CS and UML-SM. However, it is a good hint that runtime execution of generated code is semantically correct.

Limitation This evaluation methodology has the limitation that it is dependent on PSCS, PSSM and the Moka model execution. In particular, for event support, PSSM, at the writing moment, only has test cases for signal-events. For pseudo-states, histories are not supported. Thus, the evaluation result is limited to the current versions of the specifications.

Listing 5.9: C++ generated extended code for the model example in Fig. 5.22

```
class IncrementTest {
  IncrementDelegatee delegatee; //the reference to delegatee
  StateMachine Machine {
    InitialState Waiting{};
    State Incrementing{};
    FinalState FS1{};
    SignalEvent(StartSignal) Start;
    TransitionTable {
      ExtTransition(Waiting     , Incrementing ,Start , NULL , NULL)
      ExtTransition(Incrementing   , Incrementing ,NULL , guardT3, effectT3)
      ExtTransition(Incrementing   , FS1        ,NULL , guardT4, effectT4)
    }
  }
  int count;
  bool guardT3() {return count < 5;}
  bool guardT4() {return count == 5;}
  void effectT3() {
    count++;
    std::count << "<T3>";
  }
  void effectT4() {
    std::count << "<T4>";
  }
}
```

Threats to validity Operation behaviors in the PSCS and PSSM test suites are defined by activity diagrams and ALF while the tooling support requires fine-grained behavior defined as blocks of C++ code embedded into models. Therefore, an internal threat is that we manually re-create these tests and convert activity diagrams into programming language code.

5.6.2 Efficiency of generated code

In this section, we present the results obtained through the experiments on the efficiency of generated code to answer the following question.

> ***Research question 2:*** *Runtime performance and memory usage are undoubtedly critical in real-time and embedded systems. Particularly, in event-driven systems, the performance is measured by event processing speed. Are the performance and memory usage of code generated by our tool comparable to existing approaches?*

The objective of the evaluation is to compare the efficiency of code generated by Papyrus using the proposed code generation patterns, that extend the SWITCH/IF/ELSE pattern, with code generated by other tools, and to demonstrate that Papyrus can generate efficient

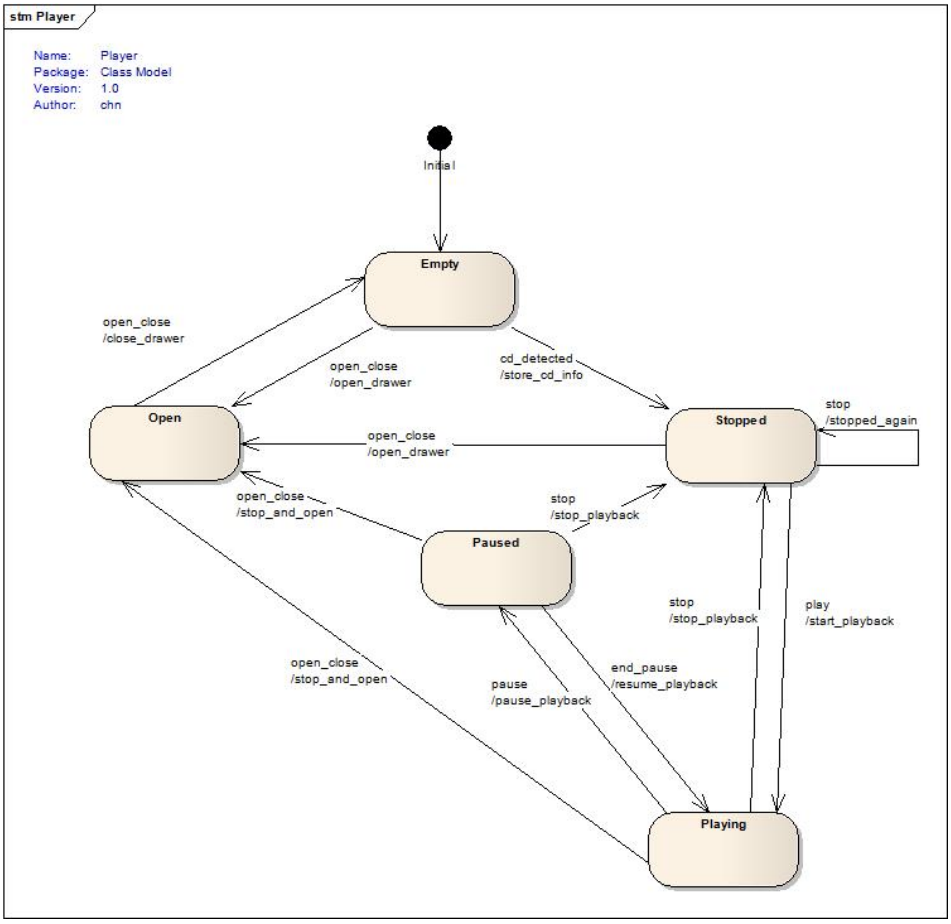


Figure 5.23: Simple state machine example from [Boost 2016b]

code. We first present the state machine examples and the tools used in the evaluation. Then, we present the comparison of the event processing performance and the memory usage of the code generated by Papyrus and the used tools.

State machine examples Two state machine examples are obtained by the preferred benchmark used by the Boost C++ libraries [Boost Library 2016a] in [Jusiak 2016]. One simple example in Fig. 5.23 on page 112 only consists of atomic/simple states and the other consists of both atomic/simple and composite states as in Fig. 5.24 on page 113.

Evaluated tools Two UML code generation tools and three C++ libraries, which statically encode a sub-set of state machine elements in C++, are included in the evaluation. The list of these approaches is shown in the list below. These tools and libraries are chosen based on their availability and the successful compilation of code generated by or code written in them. The three C++ libraries for state machines, namely MSM, MSM-Lite and EUML, are based on the Boost C++ library [Schäling 2011]. In case of IBM Rhapsody [IBM 2016a], we were not able to successfully compile generated code because of incompatibilities between generated code and C++ libraries provided by tool vendors.

- **Sinelabore** [SinelaboreRT]: It generates efficient code for Magic Draw [Magic 2016],

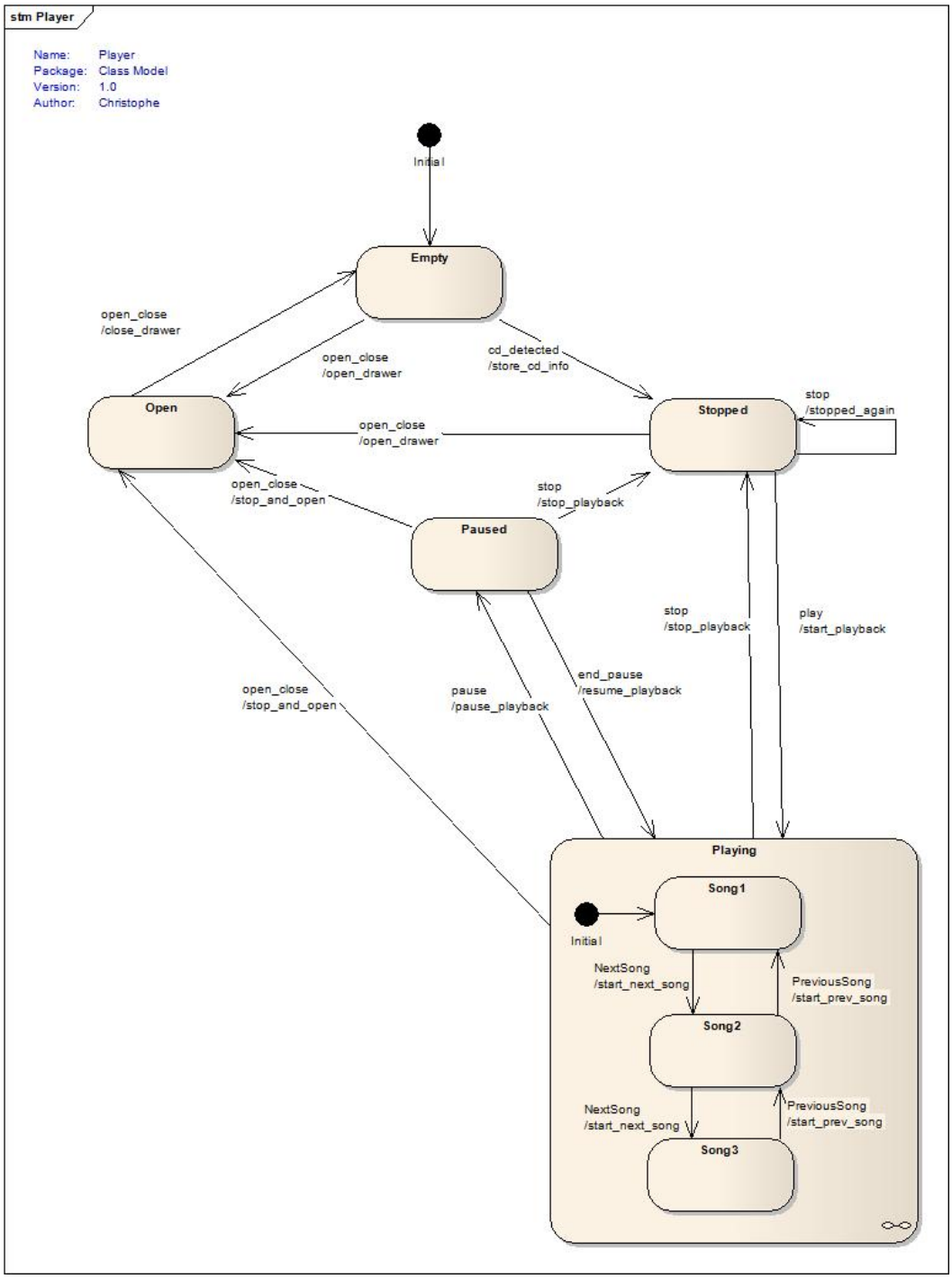


Figure 5.24: Composite state machine example from [Boost 2016a]

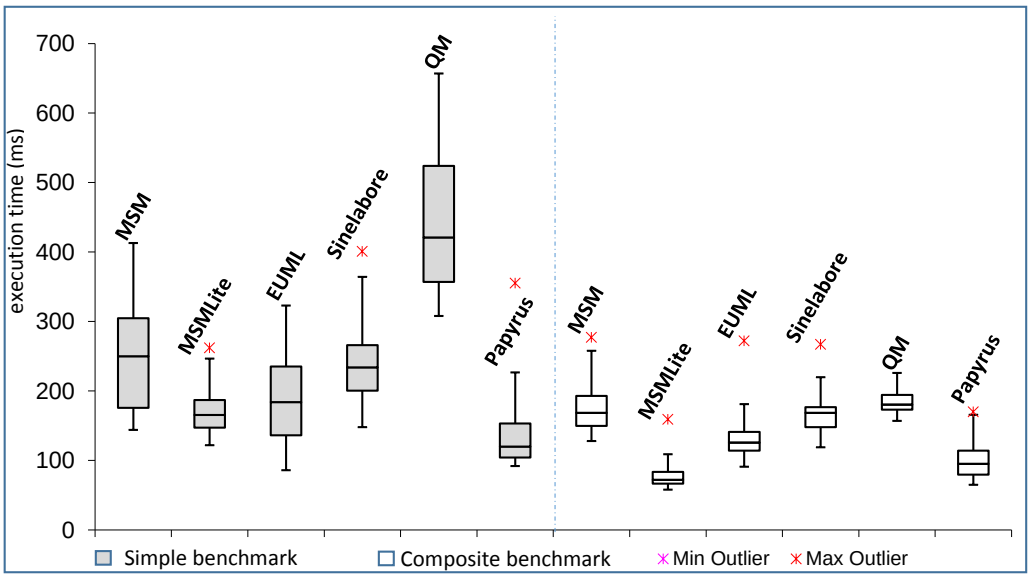


Figure 5.25: Event processing speed for the benchmarks

Enterprise Architect [SparxSysems 2016].

- **Quantum Modeling tool (QM)** [Quantum Leaps 2016]: It generates code for event-driven active object frameworks.
- **Meta State Machine (MSM)** [Boost Library 2016b]: A C++ library for a subset of state machine elements excluding signal-event, change-event, time-event, and pseudo state choice, junction, join.
- **Meta State Machine Lite (MSM-Lite)** [Jusiak 2016]: A C++14-based library for a subset of state machine elements as the MSM.
- **Function programming like-EUML** [Boost Library 2016c]: Another C++ library for the subset state machine elements as additional part of the Boost library.

Regarding the runtime environment, we used a Ubuntu virtual machine 64 bit hosted by a Windows 7 machine. For each tool, we created two applications corresponding to the two examples, generated C++ code and compiled it with GCC optimization options -O2 -s. As the experimentation conducted in [Jusiak 2016], 11 millions of events are generated and processed by the simple example and more than 4 millions for the composite example. For performance evaluation, event processing execution time is measured for each case.

Performance Fig. 5.25 shows the event processing performance of the approaches for the two benchmarks. In both of the simple and composite benchmarks, MSMLite and Papyrus run faster than the others in the scope of the experiment. Let's look closer at the event processing performance in terms of time medians. Fig. 5.26 shows the figures of the two benchmarks, relative to the performance of Sinelabore (normalized to 100%). For the simple (blue) benchmark, Papyrus (51.3%) is the fastest. For the composite (red) benchmark, with the support of C++14, the performance in MSM-Lite (42.7%) is the fastest and Papyrus is the second.

Table 5.5: Executable size in KB

State machine example	MSM	MSM-Lite	EUML	Sinelabore	QM	Papyrus
Simple	22,9	10,6	67,9	10,6	16,6	10,6
Composite	31,1	10,9	92,5	10,6	21,5	10,6

Table 5.6: Runtime memory consumption in KB. Columns from left to right are MSM, MSM-Lite, EUML, Sinelabore, QM, and Our tool, respectively.

State machine example	MSM	MSM-Lite	EUML	Sinelabore	QM	Papyrus
Composite	75.5	75.8	75.5	75.8	75.7	76.38

Memory usage Table 5.5 shows the executable size compiled from the code generated from the examples. MSMLite, Sinelabore and Papyrus require less static memory than the others. We also examine the runtime memory consumption of the composite example in the tools, we use the Valgrind Massif profiler [Valgrind 2016, Nethercote 2007] to measure memory usage. Table 5.6 shows the memory consumption measurements including stack and heap usage for the composite example. Compared to the others, the code generated by Papyrus requires a slight overhead with regard to runtime memory usage (0.35KB). This is predictable since a major part of the overhead is used for C++ multi-threading based on POSIX Threads and resource control using POSIX Mutex and Condition. However, the overhead is small and acceptable (0.35KB).

Based on the evaluation results for the performance and memory consumption of code generated by Papyrus, we assess that the proposed code generation pattern for state machine elements provides efficient code.

5.6.3 Case study

This section presents the application of the synchronization approach to the development of a relatively complex case study. The objective is to evaluate the feasibility of the approach and the correctness of the synchronization of architecture model and extended code, using the bidirectional mapping and the model-code synchronization methodological pattern. Three cases for the synchronization correctness are to be dealt with:

- Can the extended code be used to reconstruct the original architecture model?
- If the extended code is modified, can the code modifications be propagated back to the model?
- If both extended code and model are concurrently modified, can our approach make the model and code consistent again?

Such three cases are indeed similar to the synchronization evaluation through simulations presented in 4.4. However, this time, we apply the synchronization to a real case study that uses UML-CS and UML-SM elements for design, rather than randomly generated models.

The case study is an embedded software for LEGO. The LEGO car factory consists of assembly machines for small LEGO cars used for simulating a real industrial process [CEA-LIST LISE]. It is chosen for the evaluation because it is a realistic embedded system with enough complexity.

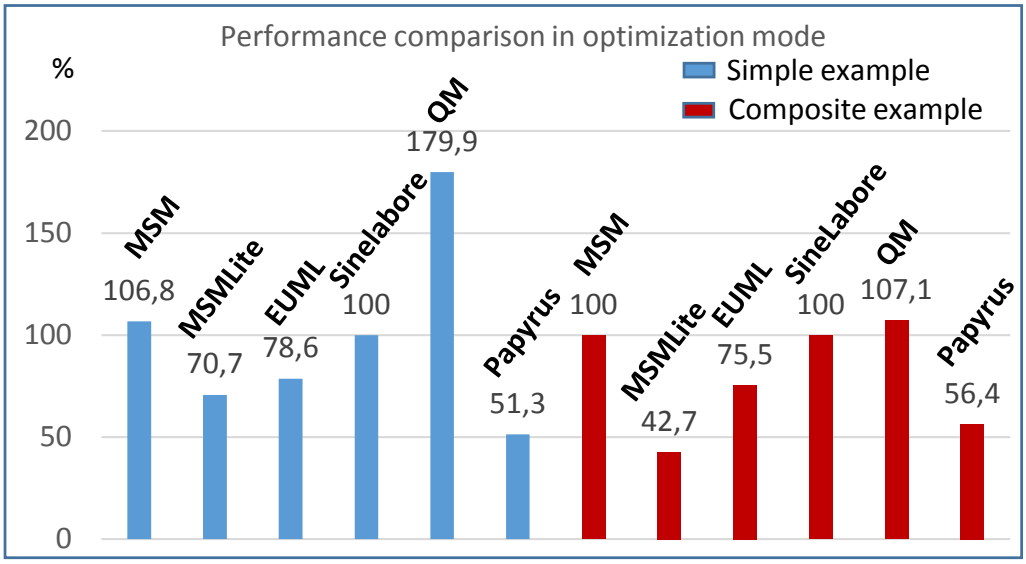


Figure 5.26: Event processing performance in optimization mode

Previously, it was developed by a developer using Papyrus without synchronization support. However, the developer found it difficult and error prone to write fine-grained code within a limited textual editor supported by the tool. The developer then programmed in a C++ editor instead to use a familiar and effective editor with programming facilities such as syntax highlights and auto-completion. She then copied the code from the C++ editor back to the model (update opaque behavior) and regenerated the code. The developer considered this process inefficient and prone-to-error. Furthermore, programming activities such as creation of methods are much easier in the code than in the model.

A LEGO car is composed of four modules: chassis, front, back, and roof. The communication between these modules is based on Bluetooth and master-slave like. In the latter, the chassis acts as master while the other modules act as slaves. Each slave module consists of five components: bluetooth communication controller, conveyor, robotic arm, press, and shelf. The behavior of each component is described by a UML-SM. The components communicate with each other through a data flow-like communication schema. Previously, each component holds references (pointers in C++), modeled as UML associations, to other components to exchange data between the components. Furthermore, the components use these references to call API of each other for interactions. The references, however, made the design not purely component-based and not reusable. Therefore, a design amelioration was studied for the synchronization case study.

To adopt a fully component-based approach, in the architecture model of the ameliorated design, we use flow ports to exchange data items/signals between the components within a module. API invocations are re-designed by using service ports. Fig. 5.27 shows the UML-CS diagram for the *front* module without showing detailed structures of each of its components. Furthermore, for simplification, only flow ports are shown in the figure. The three flow port types are used. For example, the controller can send *StopProcess* signal instances to the other four components through its ports. The robotic arm component can send the *StopProcess* signal to the conveyor or receive it from the controller through its bidirectional flow port respectively. Note that the processing of signals incoming to a

component via its ports is realized by the state machine describing the behavior of the component.

5.6.3.1 Reconstruction of model from code

In this first experimentation, for each module in the ameliorated architecture model without fine-grained code embedded as blocks of text, we generated its corresponding extended code. The model used for code generation contains 97 classes, 111 connectors, 119 ports, 15 state machines with 240 vertexes, 296 transitions and 11 events including signal-events and time-events. From the extended code, a reversed model was created by using the batch reverse engineering. This reversed model was then compared with the original module model. No differences were detected for the four modules. This result assesses that our approach and its implementation can be used for reconstructing a model from its corresponding generated code.

5.6.3.2 Propagation of code modifications back to the model

In this second experimentation, for each extended code generated, we added class attributes and fine-grained code to each component. Specifically, state actions, transition effects, and class methods are created following the rules described in Section 5.1. Furthermore, API invocations and exchange of messages/signals between components through UML associations and C++ references are re-factored with the use of service and flow ports and invocations through the required interface of the ports.

After enrichment of the generated code, we automatically propagated the code modifications back to the corresponding module model by using incremental reverse engineering. We then manually checked the module model for updates as follows:

- Are UML properties created in the model corresponding to the class attributes in the code?
- Are UML state actions and transition effects created within the model? Has each a block of text containing the fine-grained code filled in the extended code?
- Do UML operations created in the model correspond to the class methods in the code?

All of the model elements corresponding in the modified elements in the code were found in the updated model. This result assesses that our approach and its implementation can propagate modifications in code back to model.

5.6.3.3 Synchronization of concurrent modifications

To assess our approach in case of concurrent modifications, we emulated a typical situation, in which the Lego architecture model, namely, the original model, and its generated code are concurrently modified. To simplify the emulation, we only introduced modifications to method bodies in the code (become the **modified code**) and additions of UML ports, connectors and states in the model (becomes the **modified model**).

To synchronize the **modified model** and **modified code**, we used the model-code synchronization methodological pattern presented in Chapter 4 on page 45 with the specific use cases supported in Section 5.5 on page 104, whose steps are as followings:

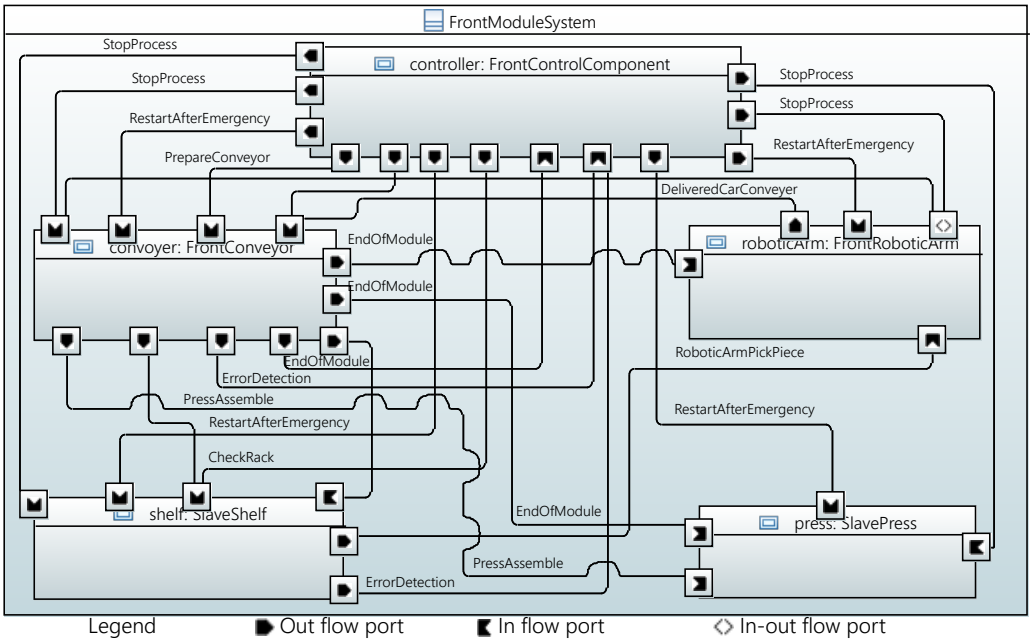


Figure 5.27: Composite structure diagram of the front module for flow ports (see Appendix E on page 142 for more details about design model and generated code.)

- Step 1: Through the incremental reverse engineering of the **modified code**, the **original model** is updated and becomes the **code-updated original model**.
- Step 2: Using EMF Compare to merge the model modifications from the **modified model** to the **code-updated original model**. The latter becomes the **synchronized model**.
- Step 3: Re-generate to create the **synchronized code** from the **synchronized model** by using batch code generation.

To assess the consistency of the **synchronized model** and **synchronized code**, we use batch reverse engineering to create a **reconstructed model**. We then compare the **reconstructed model** with the **synchronized model**. No differences were found during this comparison. We assess that our approach can synchronize architecture model and code in case of concurrent modifications.

Note that, the *model* term here means the architecture design model part that is related to code generation.

Binary size comparison After the experimentations of the synchronization, we compared the generated code in our approach (**Sync support**), that is found at [Pham 2017b], with the code generated from the original design model by Papyrus without the use of the synchronization. Table 5.7 shows the comparison of the codes generated with and without our synchronization approach. The size of the binary code created by an ARM 32 bit Linux-based cross compiler (**Sync support**) is larger than that of the approach without synchronization (**No sync**). However, the difference between the binary sizes of the executables is very subtle: our approach produces on average 6.7% larger (3% for chassis and 8% for the other modules), normalized to the binary size in case of the approach without

Table 5.7: Lego Car Factory with and without synchronization. **Sync support** (SS) denotes the code generated by the presented approach while **No sync** (NS) denotes the code generated by the existing approach

Module	Lines of code		Binary size (KB)	
	No sync	Sync support	No sync	Sync support
Chassis	9659	**11193**	256	**264**
Front	9660	**12246**	248	**268**
Roof	9725	**12232**	248	**268**
Back	9703	**12245**	268	**256**

synchronization support. The extra binary size is due to the use of component ports, that gain component-based advantages such as re-usability and independence between components, instead of associations, that tightly couple the components with each other.

Code complexity evaluation *Can the extended code reduce the program complexity compared with that of the code generated by the existing approach without synchronization support (WS)*? If the extended code is so complex that programmers cannot understand and manage, the idea of introducing additional constructs and making them executable is not significant.

Let's look at the complexity of the generated code since it should not be too complex in order for programmers to be able to understand and edit. Two metrics can be used: lines of code and cyclomatic complexity since they effectively provide means to quantitatively evaluate the complexity of a source code base [Gold 2005, Gill 1991]. The cyclomatic complexity often deals with control flow of code. However, the extended code that uses the additional constructs is based on description rather than control flow. Therefore, it is not significant to look at the cyclomatic complexity here. Hence, we only focus on lines of code to evaluate the complexity of the generated code.

The numbers of lines of code in Table 5.7 for the **Sync support** are the total of numbers of extended code and delegatee code for each module. Remember that the delegatee code is hidden from perspectives of programmers. It means that, programmers only need to manage the extended code. We then extracted the numbers of lines of the extended code to compare them with the generated code in case without synchronization. Fig. 5.28 shows the comparison results. Around 4000 lines of code are generated for delegetee code that does not need to be managed. Programmers only pay attention to the extended code whose the number of lines is around 2000 less than the code generated by the **No sync** code. Reduction of 2000 lines of code (20% of the **No sync** code) for each module would lead to significant improvement of the code complexity.

Let's look closer at what portion of the extended code programmers should manage. It is important to note that generated code for each module consists of application code and library code (this division is applied to both cases: **No sync** and **Sync support**). The application code uses API and data structures of the library code. The application code is generated from design model elements created by model-driven developers while the library code is generated from model libraries that are imported into and used by the design model of each module. It means that, the application code uses the library code but programmers do not actually need to manage the library code as well as model-driven developers do not need to manage the model libraries.

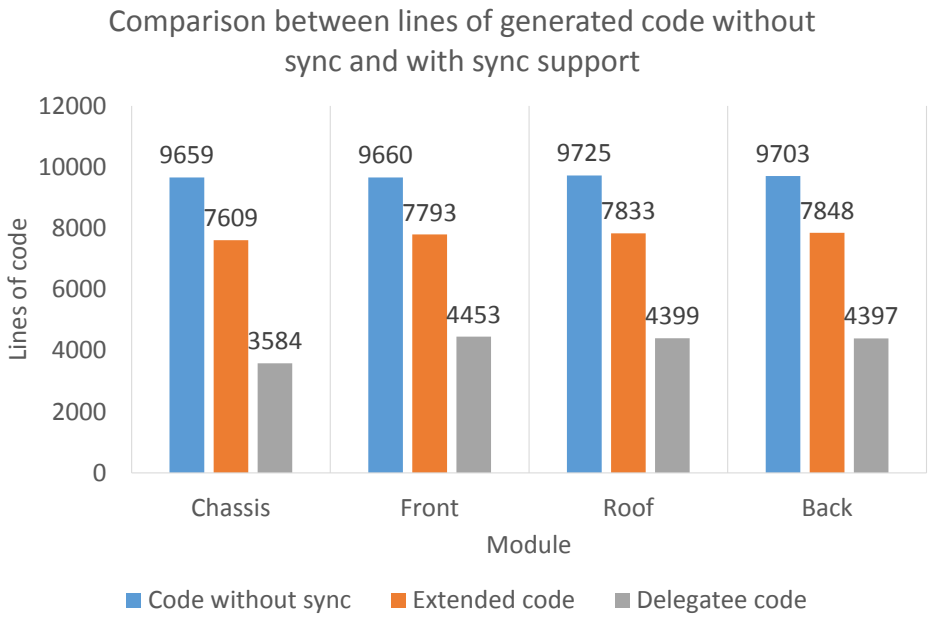


Figure 5.28: Comparison of lines of code

Now, if we only take the application code into account, only code metrics related to the main components, namely *bluetooth communication controller, conveyor, robotic arm, press* and *shelf*, of each module need to be considered. We measure the lines of the extended code of the components for each module and compare them with those of the **No sync** generated code. We measure the metrics for these codes because: programmers in case of synchronization support only need to manage the extended code of the application code. The purpose is to precisely see how much the code complexity is reduced when comparing the extended code with the **No sync** generated code within the application code.

Fig. 5.29 shows the comparison of the numbers of lines of application code to be managed in both cases: **No sync** and **Sync support**. When programming with the extended code, programmers monitor around 2000 lines of code (around 65%) less than doing with the **No sync** code.

This assesses that the proposed approach does not only bring the ability to synchronize UML-based architecture models and code, but also improves code complexity management in terms of lines of code.

There is a threat to the validity of the correctness evaluation of the synchronization based on the case study. Even though the case study is realistic, it does not cover all UML-CS and UML-SM elements. To give more confidence about the approach, we think that further evaluation based on different simulations similar to those of Chapter 4 on page 45 should be realized to take modifications made to all element types into account. However, because both of the two simulations-based evaluations have an identical methodology, the correctness of the synchronization much depends on the implementation of the use-cases. Therefore, the simulations-based evaluation in this case should not be significant.

For demonstration of the simulation-based evaluations, we developed a configurable model generator [Pham 2017a] that automatically produces random UML models that use

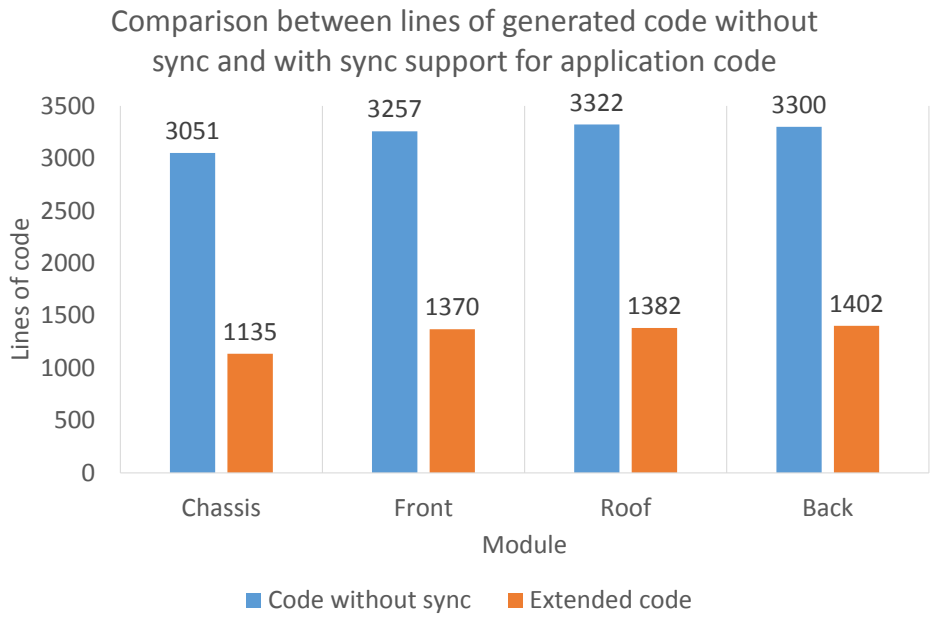


Figure 5.29: Comparison of lines of application code

UML-CS and UML-SM elements to describe software architectures and behaviors. Methodologically, each randomly generated model (original model) is then used for generating code. This latter is then used as input in a reverse engineering step to reconstruct the model. The reconstructed model and the original model are then compared. Further synchronization evaluation based on this model generator is planned for future work (see Section 6.2 on page 124 of Chapter 6 for more details about future work).

5.7 Summary

We have presented an approach for synchronization of object-oriented programming code and architecture model specified by UML-CS and UML-SM elements. The approach is based on extending an existing object-oriented programming language by introducing additional ad-hoc programming constructs for modeling concepts that have no representation in common programming languages such as Java and C++. The extended code written with the additional constructs is used as input of an in-place text-to-text transformation to generate delegatee code files to be executable. The synchronization mechanism synchronizes the extended code with the model in case of concurrent modifications.

Multiple experiments were conducted to evaluate the approach. The evaluations are tightly related to the identified requirements in Section 2.2 on page 13 for validating the contributions. Specifically, we presented the evaluation results related to the UML-conformance of code generated from model. Next, we evaluated the efficiency of generated code by using two state machine examples. Finally, the feasibility and the correctness of the approach for synchronizing UML-CS and UML-SM elements, and code are assessed through a relatively complex case study. The approach can synchronize concurrent modifications in model and code. The synchronization ability adds little memory overhead to the generated code.

Moreover, the approach allows to extract architecture model from the extended code. We think that the extracted architecture can also be used for architecture analysis. That is an other advantage of the approach: allowing to execute the system and analyze architecture.

The bidirectional mapping approach is based on using built-in programming language features to add new programming constructs to the language. Therefore, it is dependent on the programming language that we want to synchronize with UML. It is currently applicable to C++ with macros, Java and C# with annotations, and Python with macropy [Macropy 2017] (that needs to be investigated more). In future, more languages should be investigated to elaborate the approach.

Another limitation of the approach is that it does not deal with non-functional properties other than the efficiency of generated code. The non-functionalities are, for example, security as well as energy consumption. Indeed, these limitations are shared with other synchronization approaches such as *deep separation* noted in [Zheng 2012].

Chapter 6

Conclusion and perspectives

Contents

6.1 Conclusion

The hypothesis of the thesis is that developers of a complex reactive embedded software system use the UML-CS and UML-SM modeling elements for designing the system and that they use different tools for manipulating development artifacts, model and code in particular. Both design model and code can evolve concurrently since the different practices, namely modeling and programming, modify these artifacts. The problem is that, *when both design model and code are modified, it is hard to synchronize the concurrent modifications of the artifacts because there is an abstraction gap between UML-CS and UML-SM elements, and code.* Many approaches and tools have been trying to address the artifact synchronization problem in different areas such as model synchronization, bidirectional programming, model-code synchronization, round-trip engineering and co-evolution of architecture-implementation. However, these approaches fail to deal with the stated problem because of two challenges: (1) handling of the concurrent modifications of model and code; and (2) supporting the synchronization between model and code also for elements with high abstraction gap, in particular UML-CS and UML-SM elements (which have no direct representation in object-oriented programming languages).

In this thesis, a new approach for the problem of synchronization of software architecture model, specified using UML-CS and UML-SM modeling elements, and object-oriented code is developed. The proposed approach bridges the gap between the modeling and programming practices, thus allows a seamless collaboration between the developers. The approach supports the synchronization of both structural and behavioral elements of the model and the code. The approach is based on the following contributions:

1. A bidirectional mapping between an existing programming language and modeling elements by adding additional programming constructs to the existing language for modeling elements that have no representations in code.

2. A set of code generation patterns that enable the execution of the proposed constructs executable and the debugging of user-code.

3. A generic model-code synchronization methodological pattern for enabling synchronizing the extended code with the architecture model.

The approach is implemented on top of the Papyrus software designer modeling tool. Multiple experiments are conducted for evaluating the proposed approach with respect to a set of identified requirements for examining the existing approaches. In detail, the bidirectional mapping and synchronization mechanism are evaluated through multiple simulations for UML class diagram elements and C++, a case study of the Papyrus-RT runtime that is developed in C++, and a case study of the Lego car factory using UML-CS and UML-SM elements. The code generation patterns are evaluated for the completeness of the code generation support for state machine modeling elements, the semantic-conformance of the generated code (as defined in UML PSCS and PSSM): the runtime execution of generated code for each test case of the PSCS and PSSM test-suites are asserted; and the efficiency of the generated code regarding the event processing performance and the memory consumption: code generated by our pattern for two state machine examples runs fast and requires little memory consumption compared to the approaches in the scope of the thesis.

6.2 Discussion and perspectives

The content of this thesis is related to modeling, design, and implementation of software architecture, especially monolithic architectures. In this section, we discuss different perspectives that are related to the proposed approach.

Further evaluation of correctness of synchronization The evaluations of the approach presented in the thesis are related to the requirements. However, there are still issues that are not explored as shown in the following items:

- Correctness of synchronization: The correctness of the synchronization that combines the model-code synchronization methodological pattern with the bidirectional mapping is evaluated against a case study in Section 5.6. However, this evaluation only demonstrates the synchronization ability for certain element types of UML-CS and UML-SM used in the case study. Specifically, the case-study uses model elements as follows: *required* and *provided port*, *in-flow*, *out-flow*, and *bidirectional flow port*, *connector*, *state* and *state actions*, *external* and *internal transitions*, *pseudo state choice, initial*, and *signal-event* and *time-event*. Future work should take other element types into consideration to provide further evaluation.

- Learning effort measurement: The bidirectional mapping is based on introducing multiple ad-hoc additional programming constructs. Even though the latter are based on built-in language familiar features such as templates, macros and annotations to minimize learning effort of programmers, it is not clear how practical the approach is. How do programmers react to these additional constructs? Therefore, future research should elaborate an empirical study on how the constructs are used by traditional programmers. The results of this future experimentation will be able to confirm the effectiveness of the approach.

Distributed architecture A question related to our approach is whether it can be used for development of distributed architectures such as client-server architecture. Currently, it is only applied to a monolithic architecture. However, we believe that we can apply the approach to certain distributed architectures with some extensions. In our previous

work [Pham 2014], we propose to use *interaction components* to model the communication between components within a distributed system. Interaction components are first introduced in [Fredj 2010]. Connectors between components ports are transformed into interaction components. The latter are modeled as UML components/classes and then translated to a middleware-based interaction implementation (see [Pham 2014] for more details). We apply the proposed approach to synchronize code and an architecture model with interaction components in distributed architectures.

For a client-server system model as in Fig. 6.1 (a), the client requests services from the server through a remote interface. The interaction component *AsyncCall* can be split into two fragments: *client-fragment* on the client-side and *server-fragment* on the server-side. In the generated extended code in Fig. 6.1, the system at the client-side contains the client, the client-fragment, and a binding from the required port of the client to the provided port of the client-fragment. The system at the server-side, on the other hand, consists of the server, the server-fragment, and a binding from the provided port of the server and the required port of the server-fragment. The two fragments realize the communication implementation between the two sides. For example, for each call from the client through its port, the client-fragment establishes a socket connection, marshals and send the call's parameters to the server-fragment. This latter demarshals the received data to extract the parameters values sent by the client, and calls the corresponding method of the server via the required port of the interaction component.

The propagation of modifications in the model and the code is as follows:

- If the model is modified:
  - If the client is modified, propagate the modifications to the client-side code.
  - If the server is modified, propagate the modifications to the server-side code.
- If the code is modified:
  - If the client at the client-side is modified, propagate the modifications to the client component at the model level.
  - If the server at the server-side is modified, propagate the modifications to the server component at the model level.

One might ask how the propagation acts if the interaction component is modified. We argue that the generated code for the interaction component should be hidden from perspectives of the application developers because it is part of an existing library, not part of the application user-code. For separation of concerns, application developers should use interaction components to interact with other components, but not modify them. Interaction components should be developed or modified by developers of interaction components as middle-ware should be modified by middle-ware developers. Application developers can use or replace an interaction component used in the application with another one but not modify the internal code of the interaction components.

We will study how the proposed synchronization and interaction components can be used in other distributed architectures such as peer-to-peer.

Action Language for Foundational UML (ALF) Usually, UML-based MBSE approaches only generate skeleton code from UML model. To produce fully operational code,

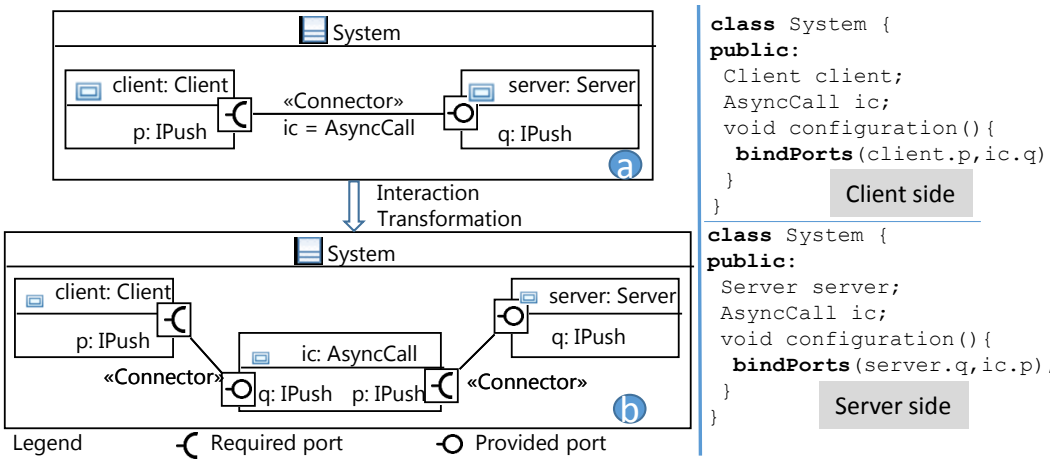


Figure 6.1: Client-server example including interaction component transformation, client-side, and server-side code

more or less additional effort is required to add more information about fine-grained behavior to the model. ALF is utilized for expression fine-grained behavior [Ciccozzi 2014]. The use of ALF enables model-based simulation execution of systems as well as fine-grained behavior code generation [Ciccozzi 2014]. However, the main limitation that makes this approach not sound is that ALF is not a common, even unknown language in the traditional programming community with very strong and widely used programming languages such as Java and C++. The question is whether users of ALF, who mainly work with models, collaborate with programmers, who use mainstream programming languages, without forcing them to change their favorite development tool and language.

In a UML design model specified by UML-CS and UML-SM elements, fine-grained behaviors are written in the bodies of the following elements, namely fine-grained behavior model elements: *operation, state actions, transition guards* and *transition effects*. In the bidirectional mapping, these elements are equivalent to class methods in code. To provide a seamless collaboration between the users of ALF, who write the bodies of the fine-grained behaviors in ALF, and programmers, who write equivalent bodies in a classic programming language, a synchronization between fine-grained behaviors based on ALF and a certain programming language should be realized in future. We believe that this synchronization is a harmonization of using ALF and common programming languages and allows to benefit from advantages of both of the modeling and programming community. Fig. 6.2 shows an example of this synchronization for an *If/Else* ALF statement written for a fine-grained behavior and an *if/else* in a C++ class method excerpted from [Ciccozzi 2016b]. In the literature, the approach in [Ciccozzi 2016b] provides a unidirectional (forward) mapping from ALF to C++. Future research should extend the approach in [Ciccozzi 2016b] for providing a backward mapping from C++ to ALF and a synchronization mechanism. The latter takes as input both of the forward and backward mapping to automate the synchronization.

Further verification of the conformance of generated code to the UML specification As discussed in Subsection 5.6.1 that presents the evaluation results related to UML-conformance of runtime execution of generated code, the evaluation has a limitation. The runtime execution of generated code only conforms to the test suites of PSCS and

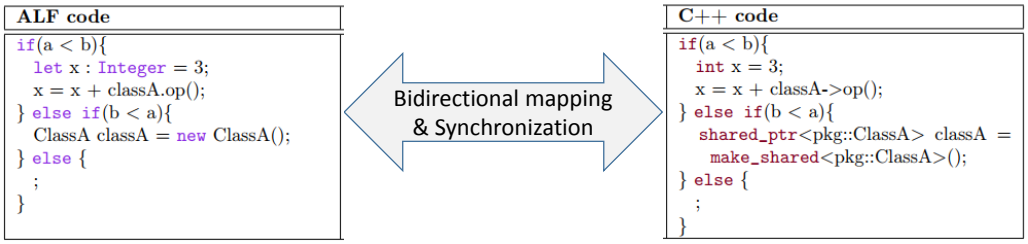


Figure 6.2: A synchronization example of ALF and C++ excerpted from [Ciccozzi 2016b]

PSSM. The results of the test suites-based evaluation are not enough to ensure that generated code always follows the standards. In the future, we will investigate model checking techniques to test certain properties of model within generated code. Furthermore, we also wish to use these model checking techniques for testing the conformance of generated code for time-events and change-events. These two events present challenges because they involve monitoring changes of some data and timing constraints. Eventually, we want to give developers more confidence about the runtime execution of generated code.

Synchronization of architecture design model and architecture analysis model In Section 1.3, we assume the partition of a UML model into two parts: code generation-related part including model elements such as UML-CS and UML-SM elements, and the other model part, namely *model analysis part*, that may be used for model-based analysis activities such as security or performance analysis. The two parts use different concepts to represent their concerns. Obviously, the two parts sometimes share the same model elements, e.g. a component used for code generation may be annotated by some UML stereotypes used for model-based analysis. In any case, the two parts always share some information that needs to be synchronized if modifications are made in one of the two parts. We think that, the synchronization of the two parts in certain cases, where model elements for model-based analysis can be produced from UML-CS and UML-SM, and vice versa, can potentially provide a further collaboration between domain experts, who do model-based analysis, designers, who mainly work with UML-CS and UML-SM elements, and programmers, who work at the code level. There is evidence that component-based design can be transformed into other models [Koziolek 2008, Petriu 2004, Petriu 2007, Xu 2003, Brosig 2011] for performance analysis. In the future, we will investigate these approaches to see whether an automated synchronization of the design model and performance analysis model is realizable.

Appendix A

Appendix: Subset of UML metamodel elements supported by the synchronization

Table A.1: UML Class and Composite Structure metamodel elements supported by the synchronization

UML diagram type	UML metamodel element
UML Class diagram	Class
	DataType
	PrimitiveType
	Property
	Operation
	OpacqueBehavior
	ValueSpecification
	TemplateSignature
	TemplateParameter
	TemplateBinding
	Interface
	Enumeration
	Package
	Usage
	Generalization
	InterfaceRealization
	Signal
UML Composite Structure diagram	Part
	Connector
	Port (with a provided and/or a required interface
	Port (in/out/inout flow ports)

Appendix B

Appendix: Generic model-code synchronization methodological pattern

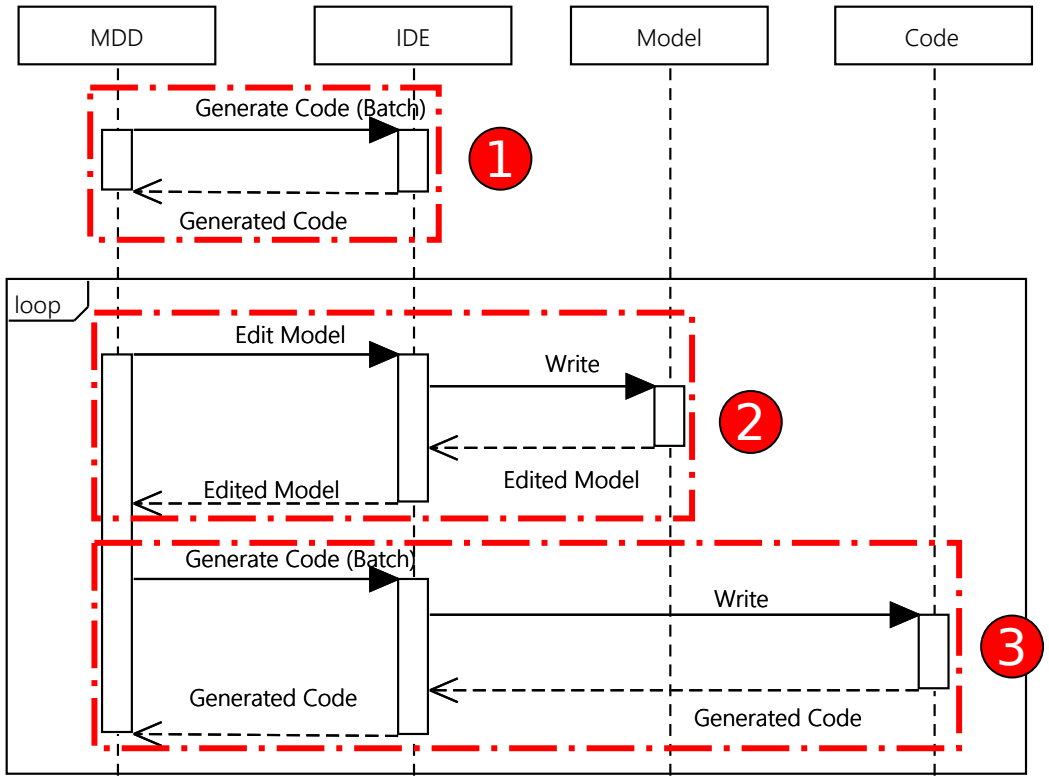


Figure B.1: Synchronization process for scenario 2, in which only code is edited (MDD = Model-Driven Developer). The API calls for Model and Code are represented generically as "Read" and "Write".

Appendix C

Appendix: Bidirectional mapping

Table C.1 presents the equivalences of two types of connectors (port-port and port-part) combining with the connector patterns (array and star), and the corresponding binding and its parameters. It is worth noting that the multiplicities of pA, a, pB and b must satisfy the conditions for corresponding connector pattern specified in the 5.1, 5.2, 5.3, and 5.4 conditions. In Table C.1, for a delegate binding, e.g. the delegate binding from pA of a container a to its inner part, the first parameter becomes *this->pA*.

Table C.1: Equivalences between connector and binding

Connector	Binding			
	Param 1	Param 2	Param 3	Param 4
Between port pA of a and pB of b- Array pattern	a.pA	b.pB	1	1
Between port pA of a and pB of b - Star pattern	a.pA	b.pB	$mul(a) * mul(pA)$	$mul(b) * mul(pB)$
Between port pA of a and part b - Array pattern	a.pA	b	1	1
Between port pA of a and part b - Star pattern	a.pA	b	$mul(a) * mul(pA)$	$mul(b)$

Appendix D

Appendix: Patterns for code generation from UML-SM elements and examples

D.1 Code generated for doActivityThread method

Listing D.1 shows a code segment for the method *doActivityThread*. The latter takes as input a state id to use and call the appropriate mutex and *doActivity*, respectively. The method does nothing and stays in a waiting point if the state corresponding to the input parameter state identifier is inactive (line 5). If the state is active, a start signal is sent to this thread method to start the execution of *doActivity*. The generated code typically follows the common paradigm in POSIX threads [Butenhof 1997].

Listing D.1: Example code generated for doActivity

```
1  while(true) {
     pthread_mutex_lock(&mutex[stateId]);
3    while(!isStarts[stateId]) {
       //await start signal
5      pthread_cond_wait(&cond, &mutex[stateId];}
     doActivity(stateId);
7    isStarts[stateId] = false;//reset wait flag
     pthread_mutex_unlock(&mutex[stateId]);
9    if (!isStops[stateId]) {
       if(stateId==IDLE_ID||...){//atomic states
11       pushCompletionEvent(stateId);
       }
13   }
   }
```

D.2 Code generation for regions

The entering method controls how a region r is entered from an outside transition and the exiting method exits completely a region by executing exit actions of sub-states from innermost to outermost.

To illustrate the entering method, we use an example as in Fig. 5.17 on page 98 with *S1* as a target composite state. *t1* is in the way (1) while *t2, t5, t6* in the way 2.

The entering method associated with the region of *S1* has a parameter *enter_mode* telling how the entering should be executed. *enter_mode* takes values depending on the number of transitions coming to the composite state. The detail of implementing these modes depends on specific programming languages that code is generated for. Listing D.2 shows the generated C++ code.

By default, the region's active sub-state is set after the execution of any effect associated with the initial transition. Therefore, $S3$ is set as active sub-state of $S1$. Entering at ($S2$)

Listing D.2: Example code generated for the region of S1

```
void S1Region1Enter(int enter_mode){
  if (enter_mode == DEFAULT) {
    states[S1_ID].actives[0] = S3_ID;
    entry(S3_ID);  sendStartSignal(S3_ID);
    S3Region1Enter(DEFAULT);
  } else if (enter_mode == S2_MODE) {
    //..
  } if (enter_mode == SH_MODE) {
    StateIDEnum his;
    if (states[S1_ID].previousActives[0]!=STATE_MAX){
      his=states[S1_ID].previousActives[0];
    } else {
      his = S2_ID;
    }
    states[S1_ID].actives[0] = his;
    entry(his);  sendStartSignal(his);
    if (S3_ID == his) {
      S3Region1Enter(S3_REGION1_DEFAULT);
    }
  } else if (enter_mode == S4_MODE) {
    states[S1_ID].actives[0] = S3_ID;
    entry(S3_ID);  sendStartSignal(S3_ID);
    S3Region1Enter(S4_MODE);
  } else if (enter_mode == ENP_MODE) {...}
}
```

sets the active sub-state of *S1* directly to *S2*. In case of an indirect sub-state ($S4$), the entry action of $S3$ is executed before $S4$ is set as the active-sub state of $S3$ and the entry execution of $S4$. It is worth noting that after the execution of each entry action, a start signal is sent to activate the waiting thread associated with *doActivity* of the corresponding state.

The method generated for exiting a region is simpler than that of entering because tt basically executes the exit actions of all the active sub-states from innermost, specified by the current active sub-state, to outermost.

D.3 Code generation for transitions and pseudo states

To discuss the implementation pattern, let $T_{trig}(e)$ be a list of transitions triggered by an e event, and $S_{trig}(e)$ be the set of source states of the transitions in $T_{trig}(e)$. $S_{trig}(e)$ is ordered based on the depth of the source states of the transitions (from innermost to outermost). We now present how to generate the code for these transitions. Algorithm 3 describes the procedure to generate the code.

The algorithm starts by finding the innermost source states of $S_{trig}(e)$. This is to ensure that the code for the transitions outgoing from the innermost states will be generated, and thus executed before that of the transitions outgoing from the outermost states. For each transition from an innermost state, code for active states and deferred events, guard checking, and transition code segments are generated by GEN_CHECK, $GEN\_GUARD(t)$ and GEN_TRANS, respectively. If the identifier of the e event is equal to one of the list of deferred events declared by the corresponding state, GEN_CHECK generates code, which checks whether should be deferred and - if yes - pushes the event to a deferred event queue managed by the runtime main thread.

For a transition t, GEN_CHECK can generate single or multiple active state checking code. The latter occurs if the target of the transition is the pseudo state join because the transitions incoming to a *join* are fired if and only if all of their source states are active. In Listing D.3, lines 2-3 show a portion of the code with multiple checking generated for the

Algorithm 3 Code generation for events

Require: Event e
Ensure: Code generation process for event method
1: **procedure** EventGenProcess(e)
2: **for** $\forall$ s $\in S_{trig}(e)$ **do**
3: $T_s = \{t \in T_{trig}(e) | src(t) = s\}$
4: **for** $\forall t \in T_s$ **do**
5: $GEN\_CHECK(s, t, e)$
6: $GEN\_GUARD(t)$
7: $GEN\_TRANS(s, t, tgt(t))$

completion event processing. The transitions *t14* and *t15* incoming to $Join1$ are executed if *S6* and *S7* are active. In addition, the code portion checks the state associated with the current completion event emitted upon the completion of either *S6*'s or *S7*'s *doActivity*. In lines 4-6, the code concurrently exits the sub-states of *S6* by using *FORK* and *JOIN*, which are respectively used to spawn and wait for a thread, for the region methods associated with *S6*'s orthogonal regions, which actually exit *S7* and *S8*. Then, *exit(S6)* is executed before the concurrency of transition effects *t14* and *t15* is taken into account.

GEN_ TRANS generates code for transitions between two vertexes. Algorithm 4 shows how it works.

Firstly, Algorithm 4 looks for the s_{ex} and s_{en} vertexes, that are contained in the same region. And s_{ex} and s_{en} also contain the source and target vertexes of the transition t, respectively. For example, s_{ex} and s_{en} in case of the $t3$ transition are $S0$ and $S1$ contained by the top region. If the transition t is part of a compound transition involving some *junction*s, IF-ELSE statements for junctions are generated first (as PSSM says *junction* is evaluated before any action). The composite state is exited by calling the associated exiting region methods (*FORK* and *JOIN* for orthogonal regions) in lines 4-9 and followed by the generated code of transition effects (lines 10-15). If the parent state s_{en} of the target vertex v_t is a state (composite state), the associated entry is executed (lines 16-18). Entering region methods are then called once the above code completes its execution (lines 19-24). If the target v_t of the transition t is a pseudo state, the generation pattern corresponding to the pseudo-state types is called. These patterns are as follows:

Note that, the procedure in 4 only applies for external transitions. Due to space limitation, the detail of generating local and internal transitions is not discussed here but the only difference is that the composite state containing the transitions is not exited.

Listing D.3: Example code generated for completion events triggering transitions t14 and t15

```
1  if(event.stateId==S6_ID || event.stateId==S7_ID){
     if(states[S6_ID].actives[0]==S7_ID &&
3       states[S6_ID].actives[1]==S8_ID) {
       thread_r1=FORK(S6Region1Exit);
5      thread_r2=FORK(S6Region2Exit);
       JOIN(thread_r1); JOIN(thread_r2);
7      sendStopSignal(S6_ID); exit(S6_ID);
       thread_t14=FORK(component->effect_t14);
9      thread_t15=FORK(component->effect_t15);
       JOIN(thread_t14); JOIN(thread_t15);
11     component->effect_t16();
       activeStateID=STATE_MAX;//inactive
13   }
   }
```

Algorithm 4 Code generation for transition

Require: A source v_s, a target vertex v_t and a transition t
Ensure: Code generation for transition

```
1: procedure GEN_TRANS(v_s, v_t, t)
2:     Find s_ex and s_en as vertexes in the same region and directly or indirectly containing/being v_s and v_t.
3:     Generate IF-ELSE statements for junctions
4:     if s_ex is a state then
5:         for r ∈ regions of s_ex do
6:             FORK(RegionExit(r))                    ▷ threads for exiting region
7:         Generate JOIN for threads created above
8:         Generate sendStopSignal to s_ex
9:         exit(s_ex)                                 ▷ exit the state
10:    if v_t is a pseudo state join then
11:        for in ∈ incoming transitions of v_t do
12:            FORK(effect(in))                       ▷ threads for transition effect
13:        Generate JOIN for threads created above
14:    else
15:        effect(t)                                  ▷ execute transition effects
16:    if s_en is a state then
17:        entry(s_en)                                ▷ state entry
18:        Generate sendStartSignal to s_en
19:    if s_en is a composite state then
20:        for r ∈ regions of s_en do
21:            FORK(RegionEnter(r))                   ▷ enter region threads
22:        Generate JOIN for threads created above
23:    else
24:        Generate for pseudo states by patterns
```

Listing D.4: Example code generated for $Fork1$ and $junc$

```
if(activeRootState==S1_ID) {
  junc = 0; //transition t9 of junc
  if (guard) {junc = 1;}
  //Exit substates of S1 and S1
  component->effect_t9();
  if(junc==0) {
    component->effect_t11();
  } else {
    component->effect_t10()
  }
  FORK(component->effect_t12()); FORK(component->effect_t3());
  //JOIN...=>concurrent execution
  //Enter state S6, S7 and S8
}
```

Table D.1: Pseudo state code generation pattern

Pseudo state	Code generation pattern
join	Use GEN_TRANS for v's outgoing transition (Listing D.3, lines 4-6).
fork	Use $FORK$ and $JOIN$ for each of outgoing transitions of v (see Listing D.4, lines 11-12).
choice	For each outgoing, an $IF - ELSE$ is generated for the guard of the outgoing together with code generated by GEN_TRANS.
junction	As a static version *choice*, a *junction* is transformed into an attribute $junc_{attr}$ and evaluated before any action executed in compound transitions (see Listing D.4, lines 2-3 and 6-10). The value of $junc_{attr}$ is then used to choose the appropriate transition at the place of *junction*.
shallow history	The identifiers of states to be exited are kept in *previousActives* of *IState*. Restoring the active states using the history is exampled as in Listing D.2. The entering method is executed as default mode at the first time the composite state is entered (lines 9-19). *previousActives* is updated with the active state identifier before exiting the region containing the history.
deep history	Saving and restoring active states are done at all state hierarchy levels from the composite state containing the deep history down to atomic states. Updating *previousActives* is committed before exiting the region, which is directly or indirectly contained by a parent state, in which a *deep history* is present.
entry point	If an *entry point* has no outgoing transition, the composite state is entered by default. Otherwise said, GEN_TRANS is called to generate code for each outgoing transition.
exit point	The code for each transition outgoing from an *exit point* is generated by using GEN_TRANS. If the *exit point* has multiple incoming transitions from orthogonal regions, it is generated as a *join* to multiple-check the source states of these incomings.
terminate	The code executes the exit action of the innermost active state, the effect of the transition and destroys the state machine object.

Appendix E

Appendix: Lego Car Case study

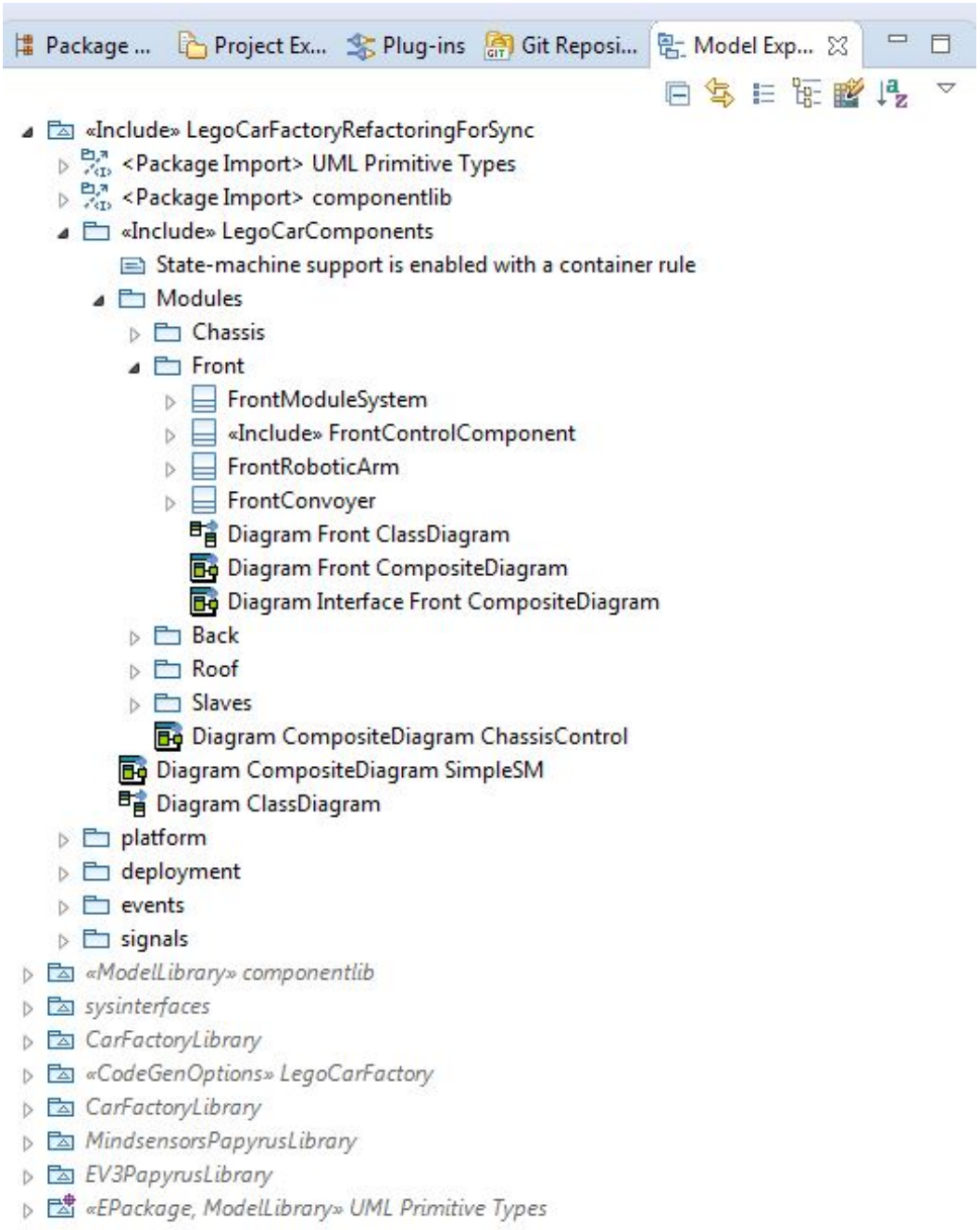


Figure E.1: The organization of the Lego Car application model contains 4 UML packages for 4 modules.

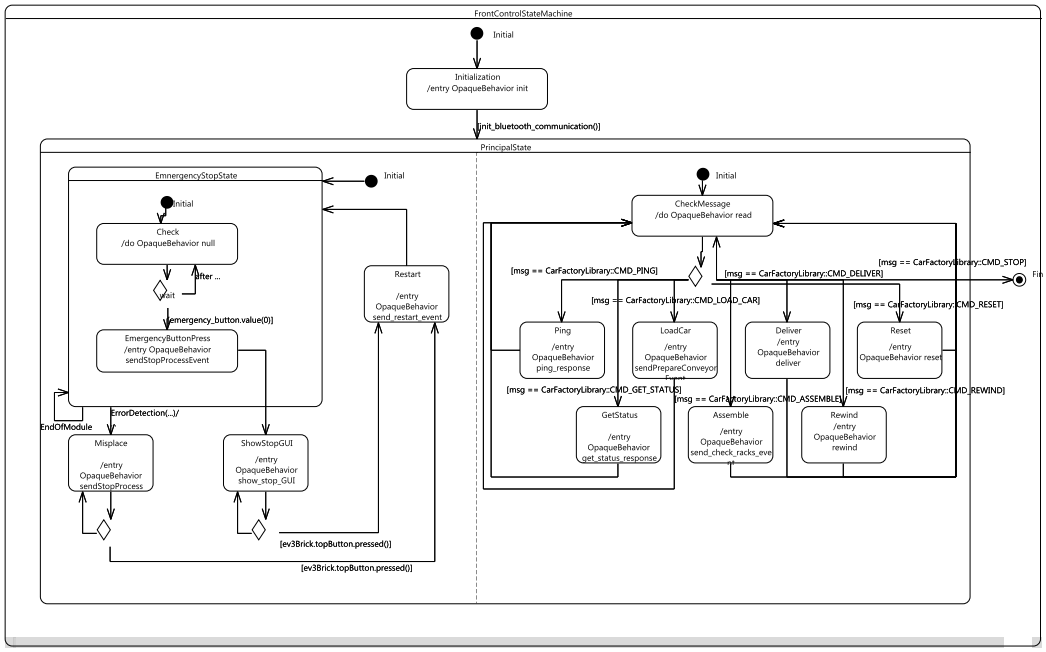


Figure E.2: The state machine diagram of the FrontControlComponent.

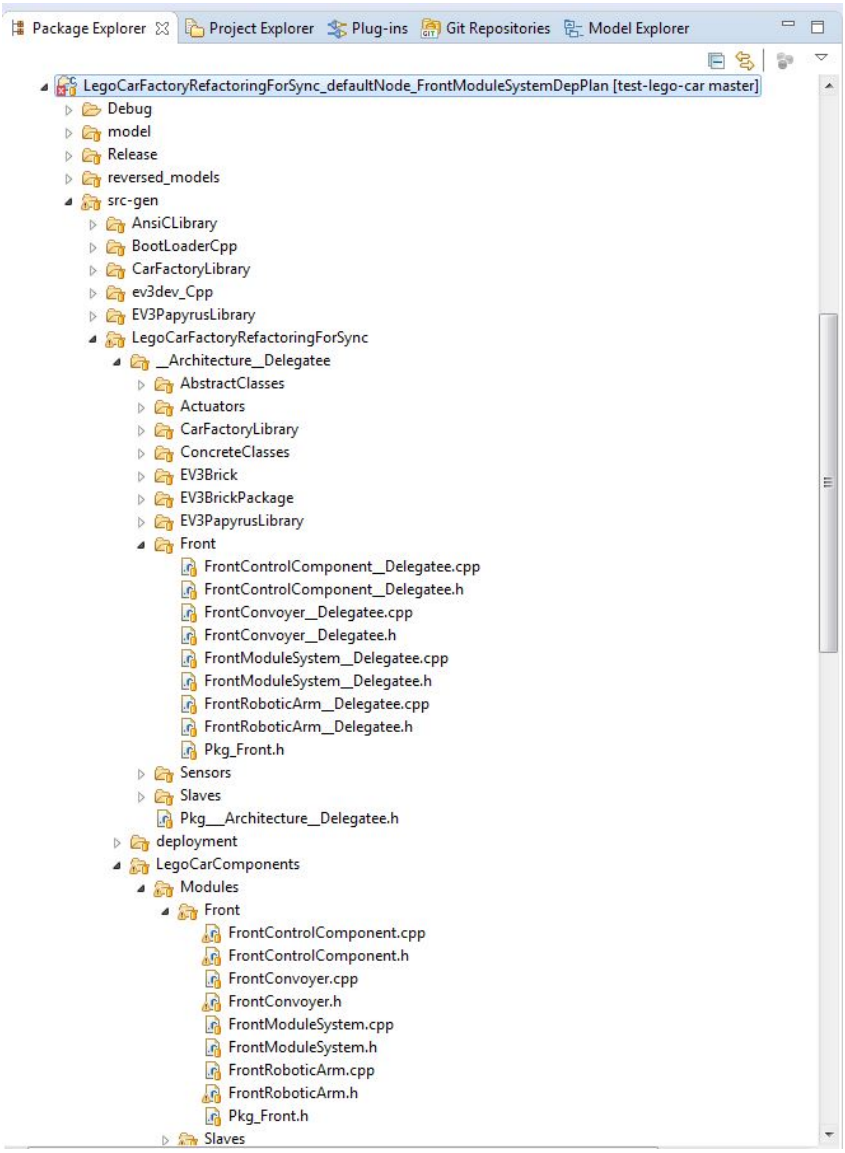


Figure E.3: The CDT C++ project generated for the front module. It contains multiple folders for C++ libraries. The "Architecture Delegatee" contains delegatee code. The "LegoCarComponents" contains extended code for the front module application model.

Listing E.1: Code segment generated for the Front module configuration,float=false

```
void FrontModuleSystem::connectorConfiguration() {
  bindPorts(controller.pOutStopProcess_Shelf, shelf.pInStopProcess);
  bindPorts(controller.pOutStopProcess_RoboticArm, roboticArm.pStopProcess);
  bindPorts(convoyer.pInStopProcess, controller.pStopProcess_Convoyer);
  bindPorts(controller.pOutStopProcess_Press, press.pInStopProcess);
  bindPorts(controller.pOutRestart_Shelf, shelf.pInRestart);
  bindPorts(controller.pOutRestart_Convoyer, convoyer.pRestart);
  bindPorts(controller.pOutRestart_Robotic, roboticArm.pInRestart);
  bindPorts(controller.pOutRestart_Press, press.pInRestart);
  bindPorts(controller.pPrepare, convoyer.pPrepare);
  bindPorts(controller.pCheckRack, shelf.pCheckRack);
  bindPorts(controller.pDelivered, convoyer.pDelivered);
  bindPorts(roboticArm.pDelivered, convoyer.pDelivered);
  bindPorts(roboticArm.pStopProcess, convoyer.pInStopProcess);
```

```
    bindPorts(convoyer.pCheckRack, shelf.pCheckRack);
16  bindPorts(convoyer.pErrDetect, controller.pErrDetect);
    bindPorts(convoyer.pEndOfMo_Control, controller.pEndOfMo);
18  bindPorts(convoyer.pEndOfMo_Shelf, shelf.pEndOfMo);
    bindPorts(convoyer.pEndOfMo_Robotic, roboticArm.pEndOfMo);
20  bindPorts(convoyer.pEndOfMo_Press, press.pEndOfMo);
    bindPorts(shelf.pErrDetect, controller.pErrDetect);
22  bindPorts(shelf.pPickPiece, roboticArm.pPickPiece);
    bindPorts(convoyer.pOutAssemble, press.pPressAssemble);
24  bindPorts(controller.pLCD, roboticArm.pLCD);
    bindPorts(controller.pLCD, shelf.pLCD);
26  bindPorts(controller.pLCD, press.pLCD);
    bindPorts(controller.pLCD, convoyer.pLCD);
28  bindPorts(controller.pModule, roboticArm.pModule);
    bindPorts(controller.pModule, convoyer.pModule);
30  bindPorts(controller.pModule, shelf.pModule);
    bindPorts(controller.pModule, press.pModule);
32  bindPorts(controller.pIFloatMotor, roboticArm.pIFloatMotor);
    bindPorts(controller.pILargeMotor, convoyer.pILargeMotor);
34  bindPorts(controller.pPressILargeMotor, press.pILargeMotor);
    bindPorts(convoyer.pGotoProcess, roboticArm.pGotoProcess);
36 }
```

Listing E.2: Code segment generated for the FrontControlComponent header,float=false

```
  class FrontControlComponent :
2 public ::CarFactoryLibrary::Module
  {
4 public:
    DECLARE_DELEGATEE_COMPONENT (FrontControlComponent)
6   StateMachine FrontControlStateMachine {
      InitialState Initialization {
8       StateEntry init();
      };
10    State PrincipalState {
        Region Region1 {
12        InitialState EmnergencyStopState {
            InitialState Check {
14            StateDoActivity doActivityCheck ();
            };
16          PseudoChoice wait{};
            State EmergencyButtonPress {
18            StateEntry sendStopProcessEvent();
            };
20        };
          State Misplace {
22          StateEntry sendStopProcess();
          };
24        State ShowStopGUI {
            StateEntry show_stop_GUI();
26        };
          PseudoChoice choice1{};
28        State Restart {
            StateEntry send_restart_event();
30        };
          PseudoChoice choice2{};
32      };
        Region Region2 {
34        InitialState CheckMessage {
            StateDoActivity read ();
36        };
          PseudoChoice choice{};
38        State Ping {
            StateEntry ping_response();
40        };
          State GetStatus {
42          StateEntry get_status_response();
          };
44        State LoadCar {
            StateEntry sendPrepareConveyorEvent();
46        };
          State Assemble {
48          StateEntry send_check_racks_event();
          };
50        State Rewind {
            StateEntry rewind();
52        };
          State Deliver {
54          StateEntry deliver();
          };
56        State Reset {
            StateEntry reset();
58        };
        };
60    };
      FinalState FinalState1 {
```

```
62      };
        TimeEvent(50) TE_50_ms_{};
64      SignalEvent(CarFactoryLibrary::events::EndOfModule) EndOfModule;
        SignalEvent(CarFactoryLibrary::events::ErrorDetection) ErrorDetection;
66      TransitionTable {
          ExT(fromInitializationtoPrincipalState, Initialization, PrincipalState,
        fromInitializationtoPrincipalStateGuard, void, NULL);
68        ExT(fromEmnergencyStopStatetoEmnergencyStopState, EmnergencyStopState,
        EmnergencyStopState, NULL, EndOfModule, NULL);
          ExT(fromEmnergencyStopStatetoMisplace, EmnergencyStopState, Misplace, NULL
        , ErrorDetection, effectFromEmnergencyStopStatetoMisplace);
70        ExT(fromChecktoWait, Check, wait, NULL, void, NULL);
          ExT(fromEmergencyButtonPresstoShowStopGUI, EmergencyButtonPress,
        ShowStopGUI, NULL, void, NULL);
72        ExT(fromMisplacetoChoice2, Misplace, choice2, NULL, void, NULL);
          ExT(fromShowStopGUItoChoice1, ShowStopGUI, choice1, NULL, void, NULL);
74        ExT(fromRestarttoEmnergencyStopState, Restart, EmnergencyStopState, NULL,
        void, NULL);
          ExT(fromCheckMessagetoChoice, CheckMessage, choice, NULL, void, NULL);
76        ExT(fromPingtoCheckMessage, Ping, CheckMessage, NULL, void, NULL);
          ExT(fromGetStatustoCheckMessage, GetStatus, CheckMessage, NULL, void, NULL
        );
78        ExT(fromLoadCartoCheckMessage, LoadCar, CheckMessage, NULL, void, NULL);
          ExT(fromAssembletoCheckMessage, Assemble, CheckMessage, NULL, void, NULL);
80        ExT(fromRewindtoCheckMessage, Rewind, CheckMessage, NULL, void, NULL);
          ExT(fromDelivertoCheckMessage, Deliver, CheckMessage, NULL, void, NULL);
82        ExT(fromResettoCheckMessage, Reset, CheckMessage, NULL, void, NULL);
          ExT(fromWaittoCheck, wait, Check, NULL, TE_50_ms_, NULL);
84        ExT(fromWaittoEmergencyButtonPress, wait, EmergencyButtonPress,
        fromWaittoEmergencyButtonPressGuard, void, NULL);
          ExT(fromChoice1toRestart, choice1, Restart, fromChoice1toRestartGuard,
        void, NULL);
86        ExT(fromChoice1toShowStopGUI, choice1, ShowStopGUI, NULL, void, NULL);
          ExT(fromChoice2toMisplace, choice2, Misplace, NULL, void, NULL);
88        ExT(fromChoice2toRestart, choice2, Restart, fromChoice2toRestartGuard,
        void, NULL);
          ExT(fromChoicetoPing, choice, Ping, fromChoicetoPingGuard, void, NULL);
90        ExT(fromChoicetoGetStatus, choice, GetStatus, fromChoicetoGetStatusGuard,
        void, NULL);
          ExT(fromChoicetoLoadCar, choice, LoadCar, fromChoicetoLoadCarGuard, void,
        NULL);
92        ExT(fromChoicetoCheckMessage, choice, CheckMessage, NULL, void, NULL);
          ExT(fromChoicetoDeliver, choice, Deliver, fromChoicetoDeliverGuard, void,
        NULL);
94        ExT(fromChoicetoRewind, choice, Rewind, fromChoicetoRewindGuard, void,
        NULL);
          ExT(fromChoicetoAssemble, choice, Assemble, fromChoicetoAssembleGuard,
        void, NULL);
96        ExT(fromChoicetoFinalState1, choice, FinalState1, fromChoicetoFinalState1
        Guard, void, NULL);
          ExT(fromChoicetoReset, choice, Reset, fromChoicetoResetGuard, void, NULL);
98      };
    };
100 InFlowPort<CarFactoryLibrary::events::ErrorDetection> pErrDetect;
    InFlowPort<CarFactoryLibrary::events::EndOfModule> pEndOfMo;
102 OutFlowPort<LegoCarFactoryRefactoringForSync::signals::StopProcess>
      pOutStopProcess_Shelf;
    OutFlowPort<LegoCarFactoryRefactoringForSync::signals::StopProcess>
      pOutStopProcess_RoboticArm;
104 OutFlowPort<LegoCarFactoryRefactoringForSync::signals::StopProcess>
      pStopProcess_Convoyer;
    OutFlowPort<LegoCarFactoryRefactoringForSync::signals::
      RestartAfterEmergencyStop> pOutRestart_Shelf;
106 OutFlowPort<LegoCarFactoryRefactoringForSync::signals::
      RestartAfterEmergencyStop> pOutRestart_Convoyer;
    OutFlowPort<LegoCarFactoryRefactoringForSync::signals::
      RestartAfterEmergencyStop> pOutRestart_Robotic;
108 OutFlowPort<LegoCarFactoryRefactoringForSync::signals::PrepareConveyor>
      pPrepare;
    OutFlowPort<LegoCarFactoryRefactoringForSync::signals::StopProcess>
      pOutStopProcess_Press;
110 OutFlowPort<LegoCarFactoryRefactoringForSync::signals::
      RestartAfterEmergencyStop> pOutRestart_Press;
    OutFlowPort<CarFactoryLibrary::events::CheckRack> pCheckRack;
112 OutFlowPort<CarFactoryLibrary::events::DeliveredCarConveyor> pDelivered;
    RequiredPort<CarFactoryLibrary::CommunicationInterfaces::
      IRoboticArmFloatMotor> pIFloatMotor;
114 RequiredPort<EV3PapyrusLibrary::Interfaces::Actuators::ILargeMotor>
      pILargeMotor;
    RequiredPort<EV3PapyrusLibrary::Interfaces::Actuators::ILargeMotor>
      pPressILargeMotor;
116 //..
};
```

Bibliography

[Abadi 2012] Moria Abadi and Yishai A Feldman. *Automatic Recovery of Statecharts from Procedural Code.* In Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, pages 238–241. ACM, 2012. (Cited on page 21.)

[Abi-Antoun 2005] M. Abi-Antoun and W. Coelho. *A Case Study in Incremental Architecture-Based Re-engineering of a Legacy Application.* In 5th Working IEEE/IFIP Conference on Software Architecture (WICSA'05), pages 159–168, 2005. (Cited on pages 33 and 43.)

[Addazi 2017] Lorenzo Addazi, Federico Ciccozzi, Philip Langer and Ernesto Posse. *Towards Seamless Hybrid Graphical–Textual Modelling for UML and Profiles.* In European Conference on Modelling Foundations and Applications, pages 20–33. Springer, 2017. (Cited on pages 34 and 35.)

[Aldrich 2002] Jonathan Aldrich, Craig Chambers and David Notkin. *ArchJava: Connecting Software Architecture to Implementation.* In Proceedings of the 24th international conference on Software engineering, pages 187–197. ACM, 2002. (Cited on pages 32 and 42.)

[Angyal 2008] László Angyal, László Lengyel and Hassan Charaf. *A Synchronizing Technique for Syntactic Model-Code Round-Trip Engineering.* In Engineering of Computer Based Systems, 2008. ECBS 2008. 15th Annual IEEE International Conference and Workshop on the, pages 463–472. IEEE, 2008. (Cited on page 25.)

[Antkiewicz 2006] Michal Antkiewicz and Krzysztof Czarnecki. *Framework-Specific Modeling Languages with Round-Trip.* ACM/IEEE 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS), pages 692–706, 2006. (Cited on pages 3 and 24.)

[Atkinson 2008] Colin Atkinson, Dietmar Stoll and Philipp Bostan. *Orthographic Software Modeling: A Practical Approach to View-Based Development.* In International Conference on Evaluation of Novel Approaches to Software Engineering, pages 206–219. Springer, 2008. (Cited on page 29.)

[Badreddin 2014] Omar Badreddin, Timothy C Lethbridge, Andrew Forward, Maged Elaasar, Hamoud Aljamaan and Miguel A Garzon. *Enhanced Code Generation from UML Composite State Machines.* In Model-Driven Engineering and Software Development (MODELSWARD), 2014 2nd International Conference on, pages 235–245. IEEE, 2014. (Cited on pages 18 and 92.)

[Bancilhon 1981] François Bancilhon and Nicolas Spyratos. *Update Semantics of Relational Views.* ACM Transactions on Database Systems (TODS), vol. 6, no. 4, pages 557–575, 1981. (Cited on pages 14 and 31.)

[Barbosa 2010] Davi MJ Barbosa, Julien Cretin, Nate Foster, Michael Greenberg and Benjamin C Pierce. *Matching Lenses: Alignment and View Update.* In ACM Sigplan Notices, volume 45, pages 193–204. ACM, 2010. (Cited on pages 30 and 32.)

[Bergmann 2012] Gábor Bergmann, István Ráth, Gergely Varró and Dániel Varró. *Change-driven model transformations.* Software & Systems Modeling, vol. 11, no. 3, pages 431–461, 2012. (Cited on page 27.)

[Bergmayr 2016] Alexander Bergmayr, Hugo Bruneliere, Jordi Cabot, Jokin García, Tanja Mayerhofer and Manuel Wimmer. *fREX: FUML-based Reverse Engineering of Executable Behavior for Software Dynamic Analysis.* In Proceedings of the 8th International Workshop on Modeling in Software Engineering, MiSE '16, pages 20–26, New York, NY, USA, 2016. ACM. (Cited on page 20.)

[Bettini 2016] Lorenzo Bettini. Implementing Domain-Specific Languages with Xtext and Xtend. Packt Publishing Ltd, 2016. (Cited on pages 8 and 35.)

[Blech 2005] J O Blech and S Glesner. *Formal Verification of Java Code Generation from UML Models.* In ... of the 3rd International Fujaba Days, pages 49–56, 2005. (Cited on page 108.)

[Bohannon 2008] Aaron Bohannon, J Nathan Foster, Benjamin C Pierce, Alexandre Pilkiewicz and Alan Schmitt. *Boomerang: Resourceful Lenses for String Data.* In ACM SIGPLAN Notices, volume 43, pages 407–419. ACM, 2008. (Cited on page 30.)

[Booch 1998] Grady Booch, James Rumbaugh and Ivar Jacobson. The Unified Modeling Language User Guide, volume 3. 1998. (Cited on pages 18, 93 and 97.)

[Boost Library 2016a] Boost Library. *Boost C++.* http://www.boost.org/, 2016. [Online; accessed 04-July-2016]. (Cited on page 112.)

[Boost Library 2016b] Boost Library. *Meta State Machine.* `http://www.boost.org/doc/libs/1_59_0_b1/libs/msm/doc/HTML/index.html`, 2016. [Online; accessed 04-July-2016]. (Cited on page 114.)

[Boost Library 2016c] Boost Library. *State Machine Benchmark.* `http://www.boost.org/doc/libs/1_61_0/libs/msm/doc/HTML/ch03s04.html`, 2016. (Cited on page 114.)

[Boost 2016a] Boost. *Composite CDPlayer Example*, 2016. [Online; accessed 14-May-2016]. (Cited on pages xiv and 113.)

[Boost 2016b] Boost. *Simple CDPlayer Example*, 2016. [Online; accessed 14-May-2016]. (Cited on pages xiv and 112.)

[Brambilla 2012] Marco Brambilla, Jordi Cabot and Manuel Wimmer. Model-Driven Software Engineering in Practice. Morgan & Claypool Publishers, 2012. (Cited on pages xiii, 7, 8 and 41.)

[Brosig 2011] Fabian Brosig, Nikolaus Huber and Samuel Kounev. *Automated Extraction of Architecture-level Performance Models of Distributed Component-based Systems.* In Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering, ASE '11, pages 183–192, Washington, DC, USA, 2011. IEEE Computer Society. (Cited on page 127.)

[Brown 2015] Neil C. C. Brown, Michael Kolling and Amjad Altadmri. *Position Paper: Lack of Keyboard Support Cripples Block-Based Programming.* In Proceedings of the 2015 Blocks and Beyond Workshop, Atlanta, GA, USA, 2015. (Cited on pages 2 and 34.)

[Brunelière 2014] H. Brunelière, J. Cabot, G. Dupé and F. Madiot. *MoDisco: A model Driven Reverse Engineering Framework*. Information and Software Technology, vol. 56, no. 8, 2014. (Cited on page 20.)

[Buneman 2000] Peter Buneman, Mary Fernandez and Dan Suciu. *UnQL: a query language and algebra for semistructured data based on structural recursion*. The VLDB Journal, vol. 9, no. 1, page 76, 2000. (Cited on page 31.)

[Butenhof 1997] David R Butenhof. Programming with POSIX threads. Addison-Wesley Professional, 1997. (Cited on pages 96 and 135.)

[Cai 2000] Xia Cai, Michael R Lyu, Kam-Fai Wong and Roy Ko. *Component-based software engineering: technologies, development frameworks, and quality assurance schemes*. In Software Engineering Conference, 2000. APSEC 2000. Proceedings. Seventh Asia-Pacific, pages 372–379. IEEE, 2000. (Cited on page 10.)

[CEA-LIST LISE] CEA-LIST LISE. *LEGO Car Factory*. http://robotics.benedettelli.com/lego-car-factory/. [Online; Accessed 22-Mar-2017]. (Cited on page 115.)

[CEA 2016] CEA. *Papyrus-RT Website*. https://www.eclipse.org/papyrus-rt/, 2016. (Cited on page 63.)

[Charfi 2012] A. Charfi, C. Mraidha and P. Boulet. *An Optimized Compilation of UML State Machines*. In 2012 IEEE 15th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing, pages 172–179, April 2012. (Cited on page 97.)

[Chen 2011] Haitao Chen and Husheng Liao. *A Survey to View Update Problem*. International Journal of Computer Theory and Engineering, vol. 3, no. 1, page 23, 2011. (Cited on page 31.)

[Cheng 2001] Shang-Wen Cheng and David Garlan. *Mapping Architectural Concepts to UML-RT*. 2001. (Cited on page 19.)

[Chikofsky 1990] Elliot J. Chikofsky and James H Cross. *Reverse engineering and design recovery: A taxonomy*. IEEE software, vol. 7, no. 1, pages 13–17, 1990. (Cited on page 20.)

[Christensen 2011] Henrik Bæ rbak Christensen and Klaus Marius Hansen. *Towards Architectural Information in Implementation: NIER Track*. 2011 33rd International Conference on Software Engineering (ICSE), pages 928–931, 2011. (Cited on page 43.)

[Christian 2017] Ries Christian Benjamin. *UML 2 State Machine for C++*. https://sourceforge.net/projects/uml2stm4cpp/, 2017. [Online; accessed 01-Aug-2017]. (Cited on page 18.)

[Cicchetti 2010] Antonio Cicchetti, Davide Di Ruscio, Romina Eramo and Alfonso Pierantonio. *JTL: A Bidirectional and Change Propagating Transformation Language*. In International Conference on Software Language Engineering, pages 183–202. Springer, 2010. (Cited on pages 26 and 28.)

[Cicchetti 2016] Antonio Cicchetti, Federico Ciccozzi and Jan Carlson. *Software Evolution Management: Industrial Practices.* In Proceedings of the 10th Workshop on Models and Evolution co-located with ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems (MODELS 2016), Saint-Malo, France, October 2, 2016., pages 8–13, 2016. (Cited on page 2.)

[Ciccozzi 2014] Federico Ciccozzi. *From Models to Code and Back: A Round-trip Approach for Model-driven Engineering of Embedded Systems.* PhD thesis, 2014. (Cited on pages 2, 9, 19, 68, 82, 87 and 126.)

[Ciccozzi 2016a] Federico Ciccozzi. *Explicit Connection Patterns (ECP) Profile and Semantics for Modelling and Generating Explicit Connections in Complex UML Composite Structures.* J. Syst. Softw., vol. 121, no. C, pages 329–344, November 2016. (Cited on pages 10 and 92.)

[Ciccozzi 2016b] Federico Ciccozzi. *On the Automated Translational Execution of the Action Language for Foundational UML.* Software & Systems Modeling, pages 1–27, 2016. (Cited on pages xiv, 19, 126 and 127.)

[Conversy 2014] Stéphane Conversy. *Unifying Textual and Visual: A Theoretical Account of the Visual Perception of Programming Languages.* In Proceedings of the 2014 ACM Internation Symposium on New Ideas, New Paradigs, and Reflections on Programming & Software, Portland, OR, USA, 2014. (Cited on page 34.)

[Cuccuru 2008] Arnaud Cuccuru, Sébastien Gérard and Ansgar Radermacher. *Meaningful Composite Structures.* Model Driven Engineering Languages and Systems, pages 828–842, 2008. (Cited on page 91.)

[Cutting 2015] David Cutting and Joost Noppen. *An Extensible Benchmark and Tooling for Comparing Reverse Engineering Approaches.* International Journal on Advances in Software, vol. 8, no. 1, pages 115–124, 2015. (Cited on pages 21 and 23.)

[Czarnecki 2009] Krzysztof Czarnecki, J Nathan Foster, Zhenjiang Hu, Ralf Lämmel, Andy Schürr and James F Terwilliger. *Bidirectional Transformations: A Cross-Discipline Perspective.* In International Conference on Theory and Practice of Model Transformations, pages 260–283. Springer, 2009. (Cited on page 30.)

[Dashofy 2001] Eric M Dashofy, André Van der Hoek and Richard N Taylor. *A Highly-Extensible, XML-Based Architecture Description Language.* In Software Architecture, 2001. Proceedings. Working IEEE/IFIP Conference on, pages 103–112. IEEE, 2001. (Cited on page 33.)

[Dayal 1982] Umeshwar Dayal and Philip A. Bernstein. *On the Correct Translation of Update Operations on Relational Views.* ACM Trans. Database Syst., vol. 7, no. 3, pages 381–416, September 1982. (Cited on page 31.)

[De Silva 2012] Lakshitha De Silva and Dharini Balasubramaniam. *Controlling Software Architecture Erosion: A Survey.* Journal of Systems and Software, vol. 85, no. 1, pages 132–151, 2012. (Cited on pages 21, 32 and 40.)

[Deligiannis 2015] Pantazis Deligiannis, Alastair F. Donaldson, Jeroen Ketema, Akash Lal and Paul Thomson. *Asynchronous Programming, Analysis and Testing with State*

Machines. SIGPLAN Not., vol. 50, no. 6, pages 154–164, June 2015. (Cited on pages 103 and 108.)

[Deors 2011] Deors. *Code Generation using Annotation Processors in the Java language part 3: Generating Source Code*, 2011. [Online; accessed 06-Sept-2017]. (Cited on page 41.)

[Desai 2013] Ankush Desai, Vivek Gupta, Ethan Jackson, Shaz Qadeer, Sriram Rajamani and Damien Zufferey. *P: Safe Asynchronous Event-Driven Programming*. ACM SIGPLAN Notices, vol. 48, no. 6, pages 321–332, 2013. (Cited on pages 103 and 108.)

[Diskin 2008] Zinovy Diskin. *Algebraic Models for Bidirectional Model Synchronization*. In Proceedings of the 11th International Conference on Model Driven Engineering Languages and Systems, MoDELS '08, pages 21–36, Berlin, Heidelberg, 2008. Springer-Verlag. (Cited on page 30.)

[Diskin 2011a] Zinovy Diskin, Yingfei Xiong and Krzysztof Czarnecki. *From State- to Delta-Based Bidirectional Model Transformations: the Asymmetric Case*. Journal of Object Technology, vol. 10, pages 6:1–25, 2011. (Cited on page 30.)

[Diskin 2011b] Zinovy Diskin, Yingfei Xiong, Krzysztof Czarnecki, Hartmut Ehrig, Frank Hermann and Fernando Orejas. *From State- to Delta-Based Bidirectional Model Transformations: The Symmetric Case*. Model Driven Engineering Languages and Systems, vol. 6981, pages 304–318, 2011. (Cited on page 30.)

[Domínguez 2012] Eladio Domínguez, Beatriz Pérez, Ángel L. Rubio and María A. Zapata. *A systematic Review of Code Generation Proposals from State Machine Specifications*, 2012. (Cited on page 17.)

[Douglass 1999] Bruce Powel Douglass. Real-time UML : Developing Efficient Objects for Embedded Systems. 1999. (Cited on pages 18, 93 and 100.)

[Dunkels 2006] Adam Dunkels, Oliver Schmidt, Thiemo Voigt and Muneeb Ali. *Protothreads: Simplifying Event-driven Programming of Memory-constrained Embedded Systems*. In Proceedings of the 4th International Conference on Embedded Networked Sensor Systems, SenSys '06, pages 29–42, New York, NY, USA, 2006. ACM. (Cited on page 1.)

[Eclipse Foundation] Eclipse Foundation. *Eclipse Projects*. https://www.eclipse.org/projects/. (Cited on page 56.)

[Engelen 2010] Luc Engelen and Mark van den Brand. *Integrating Textual and Graphical Modelling Languages*. Electron. Notes Theor. Comput. Sci., vol. 253, no. 7, pages 105–120, September 2010. (Cited on page 34.)

[Eramo 2008] R. Eramo, A. Pierantonio, J. R. Romero and A. Vallecillo. *Change Management in Multi-Viewpoint System Using ASP*. In 2008 12th Enterprise Distributed Object Computing Conference Workshops, pages 433–440, Sept 2008. (Cited on pages 28 and 29.)

[Eramo 2015] Romina Eramo, Alfonso Pierantonio and Gianni Rosa. *Managing Uncertainty in Bidirectional Model Transformations*. In Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering, SLE 2015, pages 49–58, New York, NY, USA, 2015. ACM. (Cited on page 28.)

[Fecher 2005] Harald Fecher, Jens Schönborn, Marcel Kyas and Willem-Paul de Roever. *29 New Unclarities in the Semantics of UML 2.0 State Machines.* Formal Methods and Software Engineering, pages 52–65, 2005. (Cited on page 108.)

[Finkelstein 1991] Anthony Finkelstein, Jeff Kramer and Michael Goedicke. Viewpoint oriented software development. University of London, Imperial College of Science and Technology, Department of Computing, 1991. (Cited on page 29.)

[Foster 2007] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce and Alan Schmitt. *Combinators for Bidirectional Tree Transformations: A Linguistic Approach to the View-update Problem.* ACM Trans. Program. Lang. Syst., vol. 29, no. 3, May 2007. (Cited on pages 14, 30, 32 and 58.)

[Foster 2008] J Nathan Foster, Alexandre Pilkiewicz and Benjamin C Pierce. *Quotient Lenses.* In ACM Sigplan Notices, volume 43, pages 383–396. ACM, 2008. (Cited on page 30.)

[Fredj 2010] Manel Fredj, Ansgar Radermacher, Sebastien Gerard and François Terrier. *eC3M: Optimized model-based code generation for embedded distributed software systems.* In New Technologies of Distributed Systems (NOTERE), 2010 10th Annual International Conference on, pages 279–284. IEEE, 2010. (Cited on page 125.)

[Fürst 2009] Simon Fürst, Jürgen Mössinger, Stefan Bunzel, Thomas Weber, Frank Kirschke-Biller, Peter Heitkämper, Gerulf Kinkelin, Kenji Nishikawa and Klaus Lange. *AUTOSAR–A Worldwide Standard is on the Road.* In 14th International VDI Congress Electronic Systems for Vehicles, Baden-Baden, volume 62, 2009. (Cited on page 27.)

[Gérard 2010] Sébastien Gérard, Cédric Dumoulin, Patrick Tessier and Bran Selic. *Papyrus: A UML2 Tool for Domain-Specific Language Modeling.* In Model-Based Engineering of Embedded Real-Time Systems, pages 361–368. Springer, 2010. (Cited on page 106.)

[Giese 2006] Holger Giese and Robert Wagner. *Incremental Model Synchronization with Triple Graph Grammars.* In Model Driven Engineering Languages and Systems, pages 543–557. Springer, 2006. (Cited on page 48.)

[Giese 2009] Holger Giese and Robert Wagner. *From Model Transformation to Incremental Bidirectional Model Synchronization.* Software and Systems Modeling, vol. 8, pages 21–43, 2009. (Cited on pages 3 and 27.)

[Giese 2010] Holger Giese, Stephan Hildebrandt and Stefan Neumann. *Model Synchronization at Work: Keeping SysML and AUTOSAR Models Consistent.* In Graph transformations and model-driven engineering, pages 555–579. Springer, 2010. (Cited on page 27.)

[Gill 1991] Geoffrey K. Gill and Chris F. Kemerer. *Cyclomatic Complexity Density and Software Maintenance Productivity.* IEEE Trans. Softw. Eng., vol. 17, no. 12, pages 1284–1288, December 1991. (Cited on page 119.)

[Goedicke 2000] Michael Goedicke, Bettina Enders, Torsten Meyer and Gabriele Taentzer. ViewPoint-Oriented Software Development: Tool Support for Integrating Multiple

Perspectives by Distributed Graph Transformation, pages 43–47. Springer Berlin Heidelberg, Berlin, Heidelberg, 2000. (Cited on page 29.)

[Gold 2005] Nicolas E. Gold, Andrew M. Mohan and Paul J. Layzell. *Spatial Complexity Metrics: An Investigation of Utility.* IEEE Trans. Softw. Eng., vol. 31, no. 3, pages 203–212, March 2005. (Cited on page 119.)

[Greiner 2016] Sandra Greiner, Thomas Buchmann and Bernhard Westfechtel. *Bidirectional Transformations with QVT-R: A Case Study in Round-trip Engineering UML Class Models and Java Source Code.* In MODELSWARD 2016 - Proceedings of the 4rd International Conference on Model-Driven Engineering and Software Development, Rome, Italy, 19-21 February, 2016., pages 15–27, 2016. (Cited on pages 26 and 36.)

[Grönniger 2014] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler and Steven Völkel. *Textbased Modeling.* CoRR, vol. abs/1409.6623, 2014. (Cited on pages 34 and 35.)

[Grundy 1998] John Grundy, John Hosking and Warwick B Mugridge. *Inconsistency Management for Multiple-View Software Development Environments.* IEEE Transactions on Software Engineering, vol. 24, no. 11, pages 960–981, 1998. (Cited on page 29.)

[Harel] David Harel and Hillel Kugler. *The Rhapsody Semantics of Statecharts.* (Cited on pages 92, 94 and 99.)

[Harrand 2016] Nicolas Harrand, Franck Fleurey, Brice Morin and Knut Eilif Husa. *ThingML: A Language and Code Generation Framework for Heterogeneous Targets.* In Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems, pages 125–135. ACM, 2016. (Cited on page 18.)

[Heidenreich] Florian Heidenreich, Jendrik Johannes, Mirko Seifert and Christian Wende. *Closing the Gap between Modelling and Java.* Springer. (Cited on page 20.)

[Henry 2008] Christophe Henry. *Meta State Machine (MSM).* 2008. (Cited on page 103.)

[Hermann 2012] Frank Hermann, Hartmut Ehrig, Claudia Ermel and Fernando Orejas. *Concurrent Model Synchronization with Conflict Resolution Based on Triple Graph Grammars.* In Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), volume 7212 LNCS, pages 178–193, 2012. (Cited on page 27.)

[Hettel 2008] Thomas Hettel, Michael Lawley and Kerry Raymond. *Model synchronisation: Definitions for round-trip engineering.* In International Conference on Theory and Practice of Model Transformations, pages 31–45. Springer, 2008. (Cited on pages 22, 26 and 28.)

[Hettel 2009] Thomas Hettel, Michael Lawley and Kerry Raymond. *Towards Model Round-Trip Engineering: An Abductive Approach.* In Proceedings of the 2nd International Conference on Theory and Practice of Model Transformations, Zurich, Switzerland, 2009. (Cited on page 28.)

[Hettel 2010] Thomas Hettel. *Model Round-trip Engineering*. 2010. (Cited on pages 31 and 104.)

[Hidaka 2010] Soichiro Hidaka, Zhenjiang Hu, Kazuhiro Inaba, Hiroyuki Kato, Kazutaka Matsuda and Keisuke Nakano. *Bidirectionalizing Graph Transformations*. In ACM Sigplan Notices, volume 45, pages 205–216. ACM, 2010. (Cited on page 31.)

[Hidaka 2011] Soichiro Hidaka, Kazuyuki Asada, Hiroyuki Kato, Keisuke Nakano and Zhenjiang Hu. *GRACE TECHNICAL REPORTS Towards Bidirectional Transformations on Ordered Graphs*. Technical report December, 2011. (Cited on page 30.)

[Hilliard 1999] Rich Hilliard. *Using the UML for Architectural Description*. In International Conference on the Unified Modeling Language, pages 32–48. Springer, 1999. (Cited on page 2.)

[Hinkel 2015] Georg Hinkel. *Change Propagation in an Internal Model Transformation Language*. In Theory and Practice of Model Transformations - 8th International Conference, ICMT 2015, Held as Part of STAF 2015, L'Aquila, Italy, July 20-21, 2015. Proceedings, pages 3–17, 2015. (Cited on page 103.)

[Hofmann 2011] Martin Hofmann, Benjamin Pierce and Daniel Wagner. *Symmetric Lenses*. SIGPLAN Not., vol. 46, no. 1, pages 371–384, January 2011. (Cited on page 30.)

[Holt 2008] Jon Holt and Simon Perry. SysML for Systems Engineering, volume 7. IET, 2008. (Cited on page 27.)

[Hu 2011] Zhenjiang Hu, Andy Schürr, Perdita Stevens and James Terwilliger. *Bidirectional Transformation "bx" (Dagstuhl Seminar 11031)*. Dagstuhl Reports, vol. 1, no. 1, pages 42–67, 2011. (Cited on page 30.)

[Hutchinson 2014] John Hutchinson, Jon Whittle and Mark Rouncefield. *Model-Driven Engineering Practices in Industry: Social, Organizational and Managerial Factors That Lead to Success or Failure*. Science of Computer Programming, vol. 89, pages 144 – 161, 2014. (Cited on pages 1 and 2.)

[IBM Rhapsody 2017] IBM Rhapsody. *Rational Rhapsody Frameworks and Operating Systems Reference*. `ftp://ftp.software.ibm.com/software/rationalsdp/documentation/product_doc/Rhapsody/version_7-5/framework.pdf`, 2017. [Online; accessed 15-Sept-2017]. (Cited on page 94.)

[IBM 2016a] IBM. *IBM Rhapsody*, 2016. [Online; accessed 04-July-2016]. (Cited on pages 9, 18, 19, 21, 23, 24, 36, 70, 79, 92, 95, 98 and 112.)

[IBM 2016b] IBM. *Rational DOORS*. http://www-03.ibm.com/software/products/en/ratidoor, 2016. (Cited on page 34.)

[ISO 2011] May ISO. *Systems and software engineering–architecture description*. Technical report, ISO/IEC/IEEE 42010, 2011. (Cited on page 12.)

[ISO/IEC/IEEE] ISO/IEC/IEEE. *ISO/IEC/IEEE 42010: Conceptual Model*. http://www.iso-architecture.org/ieee-1471/cm/. (Cited on page 13.)

[Jouault 2006] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, Ivan Kurtev and Patrick Valduriez. *ATL: a QVT-like transformation language.* In Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications, pages 719–720. ACM, 2006. (Cited on page 27.)

[Jouault 2014] Frédéric Jouault and Jérôme Delatour. *Towards Fixing Sketchy UML Models by Leveraging Textual Notations: Application to Real-Time Embedded Systems.* In OCL 2014, volume 1285, pages 73–82, 2014. (Cited on page 34.)

[Jusiak 2016] Kris Jusiak. *State Machine Benchmark.* `https://github.com/boost-experimental`, 2016. [Online; accessed 20-Oct-2016]. (Cited on pages 112 and 114.)

[Kelly 2007] Steven Kelly and Juha Pekka Tolvanen. Domain-Specific Modeling: Enabling Full Code Generation. 2007. (Cited on pages 3 and 23.)

[Khare 2001] Rohit Khare, Michael Guntersdorfer, Peyman Oreizy, Nenad Medvidovic and Richard N Taylor. *xADL: Enabling Architecture-Centric Tool Integration with XML.* In System Sciences, 2001. Proceedings of the 34th Annual Hawaii International Conference on, pages 9–pp. IEEE, 2001. (Cited on page 80.)

[Kiegeland 2014] J Kiegeland and H Eichler. *Medini-QVT*, 2014. (Cited on page 26.)

[Klein 1999] Thomas Klein, Ulrich A. Nickel, Jörg Niere and Albert Zündorf. *From UML to Java And Back Again.* Technical report, University of Paderborn, Paderborn, Germany, 1999. (Cited on page 23.)

[Knapp 2004] Alexander Knapp. *Semantics of UML State Machines.* 2004. (Cited on page 108.)

[Koziolek 2008] Heiko Koziolek and Ralf Reussner. *A Model Transformation from the Palladio Component Model to Layered Queueing Networks.* In SPEC International Performance Evaluation Workshop, pages 58–78. Springer, 2008. (Cited on page 127.)

[Krahn 2006] Holger Krahn and Bernhard Rumpe. *Towards Enabling Architectural Refactorings through Source Code Annotations.* Modellierung 2006, pages 203–212, 2006. (Cited on page 43.)

[Kramer 2013] Max E. Kramer, Erik Burger and Michael Langhammer. *View-centric Engineering with Synchronized Heterogeneous Models.* In Proceedings of the 1st Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling, VAO '13, pages 5:1–5:6, New York, NY, USA, 2013. ACM. (Cited on page 29.)

[Kramer 2015a] Max E. Kramer. *A Generative Approach to Change-Driven Consistency in Multi-View Modeling.* In Proceedings of the 11th International ACM SIGSOFT Conference on Quality of Software Architectures, QoSA '15, pages 129–134, New York, NY, USA, 2015. ACM. (Cited on page 29.)

[Kramer 2015b] Max E Kramer, Michael Langhammer, Dominik Messinger, Stephan Seifermann and Erik Burger. *Change-Driven Consistency for Component Code, Architectural Models, and Contracts.* In Proceedings of the 18th International ACM SIGSOFT Symposium on Component-Based Software Engineering, pages 21–26. ACM, 2015. (Cited on page 104.)

[Kusel 2013] Angelika Kusel, Juergen Etzlstorfer, Elisabeth Kapsammer, Philip Langer, Werner Retschitzegger, Johannes Schoenboeck, Wieland Schwinger and Manuel Wimmer. *A Survey on Incremental Model Transformation Approaches*. In Proceedings of the 7th Models and Evolution Workshop, Miami, FL, USA, 2013. (Cited on pages 29 and 57.)

[Leone 2006] Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri and Francesco Scarcello. *The DLV System for Knowledge Representation and Reasoning*. ACM Transactions on Computational Logic (TOCL), vol. 7, no. 3, pages 499–562, 2006. (Cited on page 28.)

[LISE] LISE. *Papyrus Software Designer*. https://wiki.eclipse.org/Papyrus_Software_Designer. 2016-04-13. (Cited on page 106.)

[Luo 2016] Zhaoyi Luo and Joanne M. Atlee. *BSML-mbeddr: Integrating Semantically Configurable State-machine Models in a C Programming Environment*. Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering - SLE 2016, pages 105–117, 2016. (Cited on page 108.)

[Macropy 2017] Macropy. *Macros in Python*. https://github.com/lihaoyi/macropy, 2017. [Online; Accessed 24-March-2017]. (Cited on page 122.)

[Magic 2016] No Magic. *Magic Draw*. https://www.nomagic.com/products/magicdraw.html, 2016. [Online; accessed 14-Mar-2016]. (Cited on pages 21, 23, 24 and 112.)

[Mancoridis 1999] S. Mancoridis, B. S. Mitchell, Y. Chen and E. R. Gansner. *Bunch: A Clustering Tool for the Recovery and Maintenance of Software System Structures*. In Proceedings of the IEEE International Conference on Software Maintenance, ICSM '99, pages 50–, Washington, DC, USA, 1999. IEEE Computer Society. (Cited on page 21.)

[Maro 2015] Salome Maro, Jan-Philipp Steghöfer, Anthony Anjorin, Matthias Tichy and Lars Gelin. *On Integrating Graphical and Textual Editors for a UML Profile Based Domain Specific Language: An Industrial Experience*. In Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering, SLE 2015, pages 1–12, New York, NY, USA, 2015. ACM. (Cited on pages 34 and 35.)

[Masunaga 1984] Yoshifumi Masunaga. *A Relational Database View Update Translation Mechanism*. In Proceedings of the 10th International Conference on Very Large Data Bases, pages 309–320. Morgan Kaufmann Publishers Inc., 1984. (Cited on pages 31 and 32.)

[Matsuda 2015] Kazutaka Matsuda and Meng Wang. *Applicative Bidirectional Programming with Lenses*. SIGPLAN Not., vol. 50, no. 9, pages 62–74, August 2015. (Cited on page 30.)

[Mazanec 2012] Martin Mazanec and Ondrej Macek. *On General-purpose Textual Modeling Languages*. Citeseer, 2012. (Cited on page 75.)

[Medeiros 1986] Claudia Bauzer Medeiros and Frank Wm. Tompa. *Understanding the Implications of View Update Policies*. Algorithmica, vol. 1, no. 1, pages 337–360, Nov 1986. (Cited on page 31.)

[Murphy 2001] Gail C Murphy, David Notkin and Kevin J. Sullivan. *Software Reflexion Models: Bridging the Gap Between Design and Implementation*. IEEE Transactions on Software Engineering, vol. 27, no. 4, pages 364–380, 2001. (Cited on page 21.)

[Mussbacher 2014] Gunter Mussbacher, Daniel Amyot, Ruth Breu, Jean-michel Bruel, Betty H C Cheng, Philippe Collet, Benoit Combemale, Robert B France, Rogardt Heldal, James Hill, Jörg Kienzle and Matthias Schöttle. *The Relevance of Model-Driven Engineering Thirty Years from Now*. ACM/IEEE 17th MODELS, pages 183–200, 2014. (Cited on page 92.)

[Nethercote 2007] Nicholas Nethercote and Julian Seward. *Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation*. In ACM Sigplan notices, volume 42, pages 89–100. ACM, 2007. (Cited on page 115.)

[Niaz 2004] Iftikhar Azim Niaz, Jiro Tanaka and others. *Mapping UML Statecharts to Java Code*. In IASTED Conf. on Software Engineering, pages 111–116, 2004. (Cited on pages 18, 97 and 98.)

[Niere 2002] Jörg Niere, Wilhelm Schäfer, Jörg P. Wadsack, Lothar Wendehals and Jim Welsh. *Towards Pattern-based Design Recovery*. In Proceedings of the 24th International Conference on Software Engineering, ICSE '02, pages 338–348, New York, NY, USA, 2002. ACM. (Cited on page 21.)

[OMG 2008] OMG. *Software & Systems Process Engineering Metamodel specification (SPEM) Version 2.0*. http://www.omg.org/spec/SPEM/2.0/PDF, 2008. (Cited on page 46.)

[OMG 2011] OMG. *A UML Profile for MARTE: Modeling and Analysis of Real- Time Embedded systems*. 2011. version 1.1 formal/2011-06-02. (Cited on pages 71 and 107.)

[OMG 2015] OMG. *Precise Semantics of UML Composite Structures (PSCS)*. 2015. [OMG Document Number: formal/2015-10-02]. (Cited on pages xiv, 69, 109 and 110.)

[Papyrus-RT 2017] Papyrus-RT. *Papyrus-RT/User/User Guide/Getting Started*. `https://wiki.eclipse.org/Papyrus-RT/User/User_Guide/Getting_Started`, 2017. [Online; accessed 12-Jul-2017]. (Cited on page 71.)

[Papyrus 2016] Papyrus. *Moka Model Execution*. `https://wiki.eclipse.org/Papyrus/UserGuide/ModelExecution`, 2016. [Online; accessed 01-Nov-2016]. (Cited on pages 101, 103 and 108.)

[Paradigm 2016] Visual Paradigm. *Visual Paradigm Homepage Website*. http://www.visual-paradigm.com/, 2016. [Online; accessed 01-Sept-2015]. (Cited on page 21.)

[Pawlak 2016] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera and Lionel Seinturier. *Spoon: A Library for Implementing Analyses and Transformations of Java Source Code*. Software: Practice and Experience, vol. 46, no. 9, pages 1155–1179, 2016. (Cited on page 41.)

[Petriu 2004] Dorin Petriu and Murray Woodside. *A Metamodel for Generating Performance Models from UML Designs*. The Unified Modeling Language. Modelling Languages and Applications, pages 41–53, 2004. (Cited on page 127.)

[Petriu 2007] Dorin B Petriu and Murray Woodside. *An Intermediate Metamodel with Scenarios and Resources for Generating Performance Models from UML Designs*. Software and Systems Modeling, vol. 6, no. 2, page 163, 2007. (Cited on page 127.)

[Pham 2014] Van Cam Pham, Önder Gürcan and Ansgar Radermacher. *Interaction Components Between Components based on a Middleware*. In 1st International Workshop on Model-Driven Engineering for Component-based Software Systems (ModComp'14), 2014. (Cited on page 125.)

[Pham 2017a] Van Cam Pham. *Architecture Model Generator*. `https://github.com/phamvancam2104/Architecture-design-model-generator.git`, 2017. [Online; Accessed 27-August-2017]. (Cited on page 120.)

[Pham 2017b] Van Cam Pham. *Github Test Lego Car*. `https://github.com/phamvancam2104/test-lego-car`, 2017. [Online; accessed 09-Aug-2016]. (Cited on page 118.)

[Pilitowski 2007] Romuald Pilitowski and Anna Dereziñska. Code Generation and Execution Framework for UML 2.0 Classes and State Machines, pages 421–427. Springer Netherlands, Dordrecht, 2007. (Cited on page 18.)

[Posse 2015] Ernesto Posse. *PapyrusRT: Modelling and Code Generation (Invited Presentation)*. In Proceedings of the International Workshop on Open Source Software for Model Driven Engineering co-located with ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS 2015), Ottawa, Canada, September 29, 2015., pages 54–63, 2015. (Cited on pages 1, 2, 9, 18, 23, 71, 79, 82, 92 and 98.)

[Quantum Leaps 2016] Quantum Leaps. *Quantum Modeling*. http://www.state-machine.com/qm/, 2016. [Online; accessed 14-May-2016]. (Cited on page 114.)

[QVT OMG 2016] QVT OMG. *Meta Object Facility (MOF) 2.0 Query / View / Transformation Specification*. 2016. formal/16-06-03 (Meta Object Facility Query/View/-Transformation, v1.3). (Cited on pages 8 and 26.)

[Rabin 1959] Michael O Rabin and Dana Scott. *Finite Automata and Their Decision Problems*. IBM journal of research and development, vol. 3, no. 2, pages 114–125, 1959. (Cited on page 10.)

[Radermacher 2009] Ansgar Radermacher, Arnaud Cuccuru, Sebastien Gerard and François 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 the Eighth International Conference on Generative Programming and Component Engineering, GPCE '09, pages 127–136, New York, NY, USA, 2009. ACM. (Cited on pages 91 and 92.)

[Rahim 2010] Lukman Ab. Rahim and Jon Whittle. *Verifying Semantic Conformance of State Machine-to-Java Code Generators*. In Model Driven Engineering Languages and Systems - 13th International Conference, MODELS 2010, Oslo, Norway, October 3-8, 2010, Proceedings, Part I, pages 166–180, 2010. (Cited on page 108.)

[Ráth 2009] István Ráth, Gergely Varró and Dániel Varró. *Change-Driven Model Transformations.* In International Conference on Model Driven Engineering Languages and Systems, pages 342–356. Springer, 2009. (Cited on page 104.)

[Reactive Manifesto] Reactive Manifesto. *Reactive Manifesto.* http://www.reactivemanifesto.org/. [Online; Accessed 14-Mar-2016]. (Cited on page 1.)

[Riedl-Ehrenleitner 2014] Markus Riedl-Ehrenleitner, Andreas Demuth and Alexander Egyed. *Towards Model-and-Code Consistency Checking.* In Proceedings - International Computer Software and Applications Conference, pages 85–90, 2014. (Cited on page 26.)

[Ringert 2014] Jan Oliver Ringert, Bernhard Rumpe and Andreas Wortmann. *From Software Architecture Structure and Behavior Modeling to Implementations of Cyber-physical Systems.* arXiv preprint arXiv:1408.5690, 2014. (Cited on page 9.)

[Romero 2009] José Raúl Romero, Juan Ignacio Jaén and Antonio Vallecillo. *Realizing Correspondences in Multi-viewpoint Specifications.* In Proceedings of the 13th IEEE International Conference on Enterprise Distributed Object Computing, EDOC'09, pages 138–147, Piscataway, NJ, USA, 2009. IEEE Press. (Cited on page 29.)

[Ruiz-gonzález 2009] Daniel Ruiz-gonzález, Nora Koch, Christian Kroiss, José raúl Romero and Antonio Vallecillo. *Viewpoint Synchronization of UWE Models.* In In Proc. of MDWE 2009, 2009. (Cited on page 29.)

[Sartipi 2003] Kamran Sartipi. *Software Architecture Recovery Based on Pattern Matching.* In Proceedings of the International Conference on Software Maintenance, ICSM '03, pages 293–, Washington, DC, USA, 2003. IEEE Computer Society. (Cited on page 21.)

[Schäling 2011] Boris Schäling. The Boost C++ Libraries. Boris Schäling, 2011. (Cited on page 112.)

[Schanzer 2015] Emmanuel Schanzer, Krishnamurthi Shriram and Fisler Kathi. *Blocks Versus Text: Ongoing Lessons from Bootstrap.* In Proceedings of the 2015 Blocks and Beyond Workshop, Atlanta, GA, USA, 2015. (Cited on page 34.)

[Seifert 2011] Mirko Seifert. *Designing Round-Trip Systems by Change Propagation and Model Partitioning.* no. April 2011, pages 1–275, 2011. (Cited on page 4.)

[Selic 1994] Bran Selic, Garth Gullekson and Paul Ward. *Real-time Object Oriented Modeling and Design.* 1994. (Cited on page 19.)

[Selic 2012] Bran Selic. *What Will It Take? A View on Adoption of Model-Based Methods in Practice.* Software & Systems Modeling, vol. 11, no. 4, pages 513–526, 2012. (Cited on pages 1, 2 and 8.)

[Sen 2016] Tamal Sen and Rajib Mall. *Extracting Finite State Representation of Java Programs.* Softw. Syst. Model., vol. 15, no. 2, pages 497–511, May 2016. (Cited on page 21.)

[Sendall 2003] Shane Sendall and Wojtek Kozaczynski. *Model Transformation the Heart and Soul of Model-Driven Software Development*. Technical report, 2003. (Cited on pages 8, 9 and 22.)

[Shalyto 2006] Anatoly Shalyto and Nikita Shamgunov. *State Machine Design Pattern*. Proc. of the 4th International Conference on. NET Technologies, 2006. (Cited on pages 18, 93 and 98.)

[Shinji 2016] Kanai Shinji and Abadi Moria. *IBM Rhapshody and UML differences*. http://www-01.ibm.com/support/docview.wss?uid=swg27040251, 2016. [Online; accessed 04-July-2016]. (Cited on pages 92, 93 and 108.)

[SinelaboreRT] SinelaboreRT. *Sinelabore Manual*. http://www.sinelabore.com/lib/exe/fetch.php?media=wiki:down (Cited on pages 92 and 112.)

[Sparx Systems 2017] Sparx Systems. *Executable State Machines*. https://www.sparxsystems.com.au/resources/user-guides/simulation/executable-state-machines.pdf, 2017. [Online; accessed 15-Sept-2017]. (Cited on pages 92 and 94.)

[SparxSysems 2016] SparxSysems. *Enterprise Architect*. http://www.sparxsystems.com/products/ea/, 2016. [Online; accessed 14-Mar-2016]. (Cited on pages 18, 19, 21, 23, 24, 70, 79, 92, 95, 98 and 114.)

[SparxSystems 2016] SparxSystems. *Enterprise Architect*. http://www.sparxsystems.eu/start/home/, September 2016. [Online; accessed 20-Nov-2016]. (Cited on page 18.)

[Specification 2015] O M G Available Specification and Change Bars. *OMG Unified Modeling Language (OMG UML)*. Language, no. November, pages 1 – 212, 2015. (Cited on pages xiii, 8, 9, 10, 12, 76, 94, 95 and 98.)

[Spinke 2013] Volker Spinke. *An Object-Oriented Implementation of Concurrent and Hierarchical State Machines*. Information and Software Technology, vol. 55, no. 10, pages 1726–1740, October 2013. (Cited on pages 18, 95, 97 and 100.)

[Steinberg 2008] Dave Steinberg, Frank Budinsky, Ed Merks and Marcelo Paternostro. EMF: Eclipse Modeling Framework. Pearson Education, 2008. (Cited on page 22.)

[Stevens 2010] Perdita Stevens. *Bidirectional model transformations in QVT: semantic issues and open questions*. Software & Systems Modeling, vol. 9, no. 1, page 7, 2010. (Cited on page 26.)

[Taylor 2007] Richard N. Taylor and Andre van der Hoek. *Software Design and Architecture The Once and Future Focus of Software Engineering*. In 2007 Future of Software Engineering, FOSE '07, pages 226–243, Washington, DC, USA, 2007. IEEE Computer Society. (Cited on page 2.)

[Terra 2012] Ricardo Terra, Marco Tulio Valente, Krzysztof Czarnecki and Roberto S Bigonha. *Recommending Refactorings to Reverse Software Architecture Erosion*. In Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on, pages 335–340. IEEE, 2012. (Cited on page 21.)

[Ubayashi 2010] Naoyasu Ubayashi, Jun Nomura and Tetsuo Tamai. *Archface: A Contract Place Where Architectural Design and Code Meet Together*. In Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1, pages 75–84. ACM, 2010. (Cited on pages xiii and 33.)

[Ubayashi 2012] Naoyasu Ubayashi and Yasutaka Kamei. *Architectural Point Mapping for Design Traceability*. In Proceedings of the Eleventh Workshop on Foundations of Aspect-Oriented Languages, FOAL '12, pages 39–44, New York, NY, USA, 2012. ACM. (Cited on page 33.)

[Ungar 1987] David Ungar and Randall B Smith. Self: The Power of Simplicity, volume 22. ACM, 1987. (Cited on page 24.)

[Valgrind 2016] Valgrind. *Valgrind Massif*. http://valgrind.org/docs/manual/ms-manual.html, 2016. [Online; accessed 20-Nov-2016]. (Cited on page 115.)

[Van Der Straeten 2008] Ragnhild Van Der Straeten, Tom Mens and Stefan Van Baelen. *Challenges in Model-Driven Software Engineering*. In International Conference on Model Driven Engineering Languages and Systems, pages 35–47. Springer, 2008. (Cited on page 2.)

[Van Gurp 2002] Jilles Van Gurp and Jan Bosch. *Design Erosion: Problems and Causes*. Journal of systems and software, vol. 61, no. 2, pages 105–119, 2002. (Cited on page 21.)

[Van Paesschen 2005] Ellen Van Paesschen, Wolfgang De Meuter and Maja D'Hondt. *Self-Sync: A Dynamic Round-Trip Engineering Environment*. In Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, pages 146–147. ACM, 2005. (Cited on pages 3 and 24.)

[Visure 2016] Visure. *Visure Requirements*. http://www.visuresolutions.com/, 2016. (Cited on page 34.)

[Von Mayrhauser 1995] A. Von Mayrhauser and A.M. Vans. *Program Comprehension during Software Maintenance and Evolution*. Computer, vol. 28, no. 8, pages 44–55, August 1995. (Cited on page 34.)

[Westfechtel 2015] B. Westfechtel. *A Case Study for Evaluating Bidirectional Transformations in QVT Relations*. In 2015 International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE), pages 141–155, April 2015. (Cited on page 26.)

[Wider 2011] Arif Wider. *Towards Combinators for Bidirectional Model Transformations in Scala*. In International Conference on Software Language Engineering, pages 367–377. Springer, 2011. (Cited on page 30.)

[Woods 2010] Eoin Woods and Nick Rozanski. *Unifying Software Architecture with Its Implementation*. Proceedings of the Fourth European Conference on Software Architecture Companion Volume - ECSA '10, page 55, 2010. (Cited on page 39.)

[Xiong 2007] Yingfei Xiong, Dongxi Liu, Zhenjiang Hu, Haiyan Zhao, Masato Takeichi and Hong Mei. *Towards Automatic Model Synchronization from Model Transformations*. Proceedings of the twentysecond IEEEACM international conference on Automated software engineering ASE 07, pages 164–173, 2007. (Cited on pages 14, 15 and 27.)

[Xiong 2009] Yingfei Xiong. *A Language-based Approach to Model Synchronization in Software Engineering*. PhD Dissertation, 2009. (Cited on pages 30 and 31.)

[Xu 2003] Jing Xu, Murray Woodside and Dorina Petriu. Performance Analysis of a Software Design Using the UML Profile for Schedulability, Performance, and Time, pages 291–307. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003. (Cited on page 127.)

[Yatta Solutions 2012] Yatta Solutions. *UML Lab*. `http://www.uml-lab.com`, 2012. (Cited on pages 21 and 23.)

[Yu 2012] Yijun Yu, Yu Lin, Zhenjiang Hu, Soichiro Hidaka, Hiroyuki Kato and Lionel Montrieux. *Maintaining Invariant Traceability through Bidirectional Transformations*. 2012 34th International Conference on Software Engineering (ICSE), pages 540–550, jun 2012. (Cited on pages xiii and 25.)

[Zhai 2016] Juan Zhai, Jianjun Huang, Shiqing Ma, Xiangyu Zhang, Lin Tan, Jianhua Zhao and Feng Qin. *Automatic Model Generation from Documentation for Java API Functions*. In Proceedings of the 38th International Conference on Software Engineering, ICSE '16, New York, NY, USA, 2016. ACM. (Cited on page 40.)

[Zhang 2014] Dan Zhang, Dragan Bošnacki, Mark van den Brand, Luc Engelen, Cornelis Huizing, Ruurd Kuiper and Anton Wijs. *Towards Verified Java Code Generation from Concurrent State Machines*. In AMT 2014–Analysis of Model Transformations Workshop Proceedings, page 64. Citeseer, 2014. (Cited on page 108.)

[Zheng 2012] Yongjie Zheng and Richard N Taylor. *Enhancing Architecture-Implementation Conformance with Change Management and Support for Behavioral Mapping*. In Proceedings of the 34th International Conference on Software Engineering, pages 628–638. IEEE Press, 2012. (Cited on pages 3, 9, 22, 33, 80, 93 and 122.)

université PARIS-SACLAY
ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Titre: Model-Based Software Engineering: Methodologies pour la synchronisation entre modèle et code dans le développement de systèmes réactifs

Mots clés: machines à état, systèmes embarqués, architecture basée sur les composants, synchronisation entre model et code, génération de code, architectes de logiciels

Résumé: Model-Based Software Engineering (MBSE) a été proposé comme une méthodologie prometteuse de développement de logiciels pour surmonter les limites de la méthodologie traditionnelle basée sur la programmation pour faire face à la complexité des systèmes embarqués. MBSE favorise l'utilisation de langages de modélisation pour décrire les systèmes d'une manière abstraite et fournit des moyens pour générer automatiquement de différents artefacts de développement, p.ex. code et documentation, à partir de modèles. Le développement d'un système complexe implique souvent de multiples intervenants qui utilisent différents outils pour modifier les artefacts de développement, le modèle et le code en particulier dans cette thèse. Les modifications apportées aux artefacts évoquent le problème de cohérence qui nécessite un processus de synchronisation pour propager les modifications apportées dans l'un artefact aux autres artefacts.

Dans cette étude, le problème de la synchronisation des modèles d'architecture basés sur les éléments UML composite structure (UML-CS) et UML state machine (UML-SM) du langage de l'Unified Modeling Language (UML), et le code orienté objet est présenté. UML-CSs sont utilisés pour décrire l'architecture du logiciel basée sur les composants et UML-SMs pour les comportements discrets liés aux événements des systèmes réactifs. Le premier défi est de permettre une collaboration entre les architectes de logiciels et les programmeurs produisant de modèle et de code, en utilisant différents outils. Il soulève le problème de synchronisation où il existe de **modifications simultanées** des artefacts. En fait, il existe un écart de perception entre les langages à base de diagramme (langages de modélisation) et les langages textuels (langages de programmation). D'une part, les programmeurs préfèrent souvent utiliser la combinaison familière d'un langage de programmation et d'un environnement de développement intégré. D'autre part, les architectes logiciels, travaillant à des niveaux d'abstraction plus élevés, favorisent l'utilisation des modèles et préfèrent donc les langages à base de diagramme pour décrire l'architecture du système. Le deuxième défi est qu'il existe un **écart d'abstraction** significatif entre les éléments du modèle et les éléments du code: les éléments UML-CS et UML-SM sont au niveau d'abstraction plus élevé que les éléments du code. L'écart rend la difficulté pour les approches de synchronisation actuelles car il n'y a pas de façon facile de réflecter les modifications du code au modèle.

Cette thèse propose une approche automatisée de synchronisation composée de deux principales contributions corrélées. Pour aborder le premier défi, on propose un patron méthodologique générique de synchronisation entre modèle et code. Il consiste en des définitions des fonctionnalités nécessaires et plusieurs processus qui synchronisent le modèle et le code en fonction de plusieurs scénarios définis où les développeurs utilisent différents outils pour modifier le modèle et le code. Cette contribution est indépendante de UML-CSs et UML-SMs. La deuxième contribution traite du deuxième défi et est basée sur les résultats de la première contribution. Dans la deuxième contribution, un **mapping bidirectionnel** est présentée pour réduire l'écart d'abstraction entre le modèle et le code. Le mapping est un ensemble de correspondances entre les éléments de modèle et ceux de code. Il est utilisé comme entrée principale du patron méthodologique générique de synchronisation entre modèle et code. Plus important, l'utilisation du mapping fournit les fonctionnalités définies dans la première contribution et facilite la synchronisation des éléments de UML-CS et UML-SM et du code. L'approche est évaluée au moyen de multiples simulations et d'une étude de cas.

Université Paris-Saclay
Espace Technologique / Immeuble Discovery
Route de l'Orme aux Merisiers RD 128 / 91190 Saint-Aubin, France

université PARIS-SACLAY
ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Title: Model-Based Software Engineering: Methodologies for Model-Code Synchronization in Reactive System Development

Keywords: state machines, embedded systems, component-based architecture, model-code synchronization, code generation, software architects

Abstract: MBSE has been proposed as a promising software development methodology to overcome limitations of traditional programming-based methodology in dealing with the complexity of embedded systems. MBSE promotes the use of modeling languages for describing systems in an abstract way and provides means for automatically generating different development artifacts, e.g. code and documentation, from models. The development of a complex system often involves multiple stakeholders who use different tools to modify the development artifacts, model and code in particular in this thesis. Artifact modifications must be kept consistent: a synchronization process needs to propagate modifications made in one artifact to the other artifacts.
In this study, the problem of synchronizing UML-based architecture models, specified by UML-CS and UML-SM elements, and object-oriented code is presented. UML-CSs are used for describing the component-based software architecture and UML-SMs for discrete event-driven behaviors of reactive systems. The first challenge is to enable a collaboration between software architects and programmers producing model and code by using different tools. This raises the synchronization problem of concurrent artifact modifications. In fact, there is a perception gap between diagram-based languages (modeling languages) and text-based languages (programming languages). On the one hand, programmers often prefer to use the more familiar combination of a programming language and an Integrated Development Environment. On the other hand, software architects, working at higher levels of abstraction, tend to favor the use of models, and therefore prefer diagram-based languages for describing the architecture of the system. The second challenge is that there is a significant abstraction gap between the model elements and the code elements: UML-CS and UML-SM elements are at higher level of abstraction than code elements. The gap makes current synchronization approaches hard to be applied since there is no easy way to reflect modifications in code back to model.
This thesis proposes an automated synchronization approach that is composed of two main correlated contributions. To address the first challenge, a generic model-code synchronization methodological pattern is proposed. It consists of definitions of necessary functionalities and multiple processes that synchronize model and code based on several defined scenarios where the developers use different tools to modify model and code. This contribution is independent of UML-CSs and UML-SMs. The second contribution deals with the second challenge and is based on the results from the first contribution. In the second contribution, a bidirectional mapping is presented for reducing the abstraction gap between model and code. The mapping is a set of correspondences between model elements and code elements. It is used as main input of the generic model-code synchronization methodological pattern. More importantly, the usage of the mapping provides the functionalities defined in the first contribution and eases the synchronization of UML-CS and UML-SM elements and code. The approach is evaluated by means of multiple simulations and a case study.

Université Paris-Saclay
Espace Technologique / Immeuble Discovery
Route de l'Orme aux Merisiers RD 128 / 91190 Saint-Aubin, France