Karlsruhe Institute of Technology

Consistency Preservation in the Development Process of Automotive Software

Master's Thesis of

Manar Mazkatli

at the Department of Informatics
Institute for Program Structures and Data Organization (IPD)

Reviewer:	Jun. Prof. Anne Koziolek
Second reviewer:	Prof. Ralf Reussner
Advisor:	Dr.-Ing. Erik Burger
Second advisor:	Dr.-Ing. Jochen Quante

01. January 2016 – 30. June 2016

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

Karlsruhe, 30.06.2016

. . . .

(Manar Mazkatli)

// Abstract

The development of automotive systems is becoming more and more complex as several modeling formalisms and languages are used to describe the same system from different viewpoints. These formalisms and languages offer domain-specific analysis of the system under development using specific modeling notations and levels of abstraction. The Automotive Electronics (AE) division of Bosch-group, for example, uses SysML for system engineering, AMALTHEA platform for embedded multi- or many- core software engineering, and the ASCET product family for embedded automotive software development.

These heterogeneous models can share common semantics. That implies that changes in an artifact may require multiple updates in different models due to the fragmentation of information across the models. These models are separately developed by different project partners. Therefore, the synchronisation between them is more complex, an error-prone and may result in an inconsistency if not all related affected models are synchronized.

Currently, the consistency is preserved and reestablished using an expensive cumbersome manual process. Furthermore there is no efficient mechanism to check consistency between the models. The potential inconsistent cases are detected for the first time at a very late stage, e.g., by the linker. Besides that, resolving the conflicts at this stage is expensive and leads also to drift and erosion between the models and the implementation.

The Vitruvius approach for view-based software development provides synchronization mechanism between heterogonous models based on change-driven model transformations, which automatically propagate the change to the related artifacts. These transformations can be generated by Mapping Invariants and Responses (MIR) language from declarative expression of the correspondence rules between the metamodels. These rules define the mapping between the artifacts of the metamodels, the consistency constraints and the appropriate actions to keep the consistency when the constraints are violated. However, the current Vitruvius prototype supports neither automotive modeling tools nor the ability to integrate legacy models that are developed by external tools.

The proposed work in this thesis enables the Vitruvius approach to support the automotive system development through two contributions. The first one is identifying the correspondence rules between the SysML, ASCET, and AMALTHEA metamodels and explaining the obstacles that prevents using the current version of MIR language to declare these rules. The second contribution is extending Vitruvius to enable the integration of legacy models in the development process. For this purpose, this work proposes an algorithm for automatic consistency check and semi-automatic resolving of the potential conflicts.

This algorithm is evaluated by a case study from AE division of Bosch-group. In this case study, a control algorithm is defined using SysML, AMALTHEA and ASCET models. This evaluation shows the ability of our proposed algorithm to automatically detect the inconsistent cases between these models and solve the conflicts semi-automatically during

the design stage. As a result, the costs and the efforts of preserving the consistency are significantly reduced compared to the previous manual solutions.

Zusammenfassung

Die Entwicklung von Automobilesystemen wird immer komplexer, da mehrere Modellierungsformalismen und Sprachen verwendet werden, um das gleiche System von verschiedenen Standpunkten zu beschreiben. Diese Formalismen und Sprachen bieten domänenspezifische Analyse des Systems mittels spezifischen Modellierungsnotationen und Abstraktionsebenen.

Zum Beispiel verwendet der Bosch-Geschäftsbereich Automotive Electronics (AE) SysML für Systems Engineering, die AMALTHEA-Plattform für eingebettete Vielkern-/ Mehrkern-Softwaretechnik und die ASCET-Produktfamilie für modellbasierte Entwicklung eingebetteter Automobilsoftware. Diese heterogenen Modelle können eine gemeinsame Semantik teilen. Daher können die Änderungen in einem Artefakt aufgrund der Fragmentierung von Informationen auf mehreren Modellen zu mehreren Updates in verschiedenen Modellen führen. Darüber hinaus werden diese Modelle separat voneinander manuell von unterschiedlichen Projektpartnern gepflegt. Dies kann zur Inkonsistenz führen, wenn nicht alle durch die Änderung beeinflusste Modelle synchronisiert sind.

Momentan wird die Konsistenz mit einem teuren umständlichen manuellen Verfahren erhalten. Dazu gibt es keinen wirksamen Mechanismus, um die Konsistenz der Modelle zu prüfen. Die potentielle Inkonsistenz wird erst in einer sehr späten Phase wie beim Übersetzen/Linken festgestellt, dabei werden nicht alle Inkonsistenzen entdeckt. Außerdem ist die Konfliktlösung im dieser Phase teuer und führt auch zu Drift und Erosion zwischen den Modellen und der Implementierung.

Der Vitruvius-Ansatz für die sichtbasierte Software-Entwicklung bietet einen Synchronisationsmechanismus zwischen heterogenen Modellen an, der die Änderung automatisch dank der modelbasierten Transformationen auf die relevanten Artefakte propagiert. Diese Transformationen können durch Mapping Invarianten und Responsen (MIR) Sprache aus deklarativem Ausdruck der Korrespondenzregeln zwischen den Metamodellen generiert werden. Diese Regeln definieren die Abbildung zwischen den Artefakten der Metamodelle, die Konsistenz-Einschränkungen und die entsprechenden Maßnahmen, die die Konsistenz halten, wenn die Einschränkungen verletzt werden. Allerdings unterstützt die aktuellen Vitruvius Prototyp weder Automobil-Modellierungs-Werkzeuge noch die Fähigkeit, die von externen Tools entwickelte Legacy Modelle zu integrieren.

Durch zwei Beiträge ermöglicht es meine Masterarbeit dem Vitruvius-Ansatz, die Automobil-Systementwicklung zu unterstützen. Der erste Beitrag ist die Identifizierung der Haupt-Korrespondenzregeln zwischen den SysML, ASCET und AMALTHEA Metamodelln und Erklärung der Hindernisse, die den deklarative Ausdruck dieser Regeln sowie die automatische Generierung der Modell-Transformationen verhindern.

Der zweite Beitrag erweitert Vitruvius, um die Integration von Legacy-Modelle in den Vitruvius Entwicklungsprozess zu ermöglichen. Zu diesem Zweck stellt diese Arbeit einen

Algorithmus für die automatische Konsistenzprüfung und halbautomatische Konfliktlösung vor.

Dieses Algorithmus wird durch eine Fallstudie von AE-Bosch-Geschäftsbereich evaluiert. In dieser Fallstudie wird ein Steuer-Algorithmus mit SysML, AMALTHEA und ASCET Modelle definiert. Die Evaluation zeigt die Fähigkeit unseres vorgeschlagenen Algorithmus, die inkonsistenten Fälle zwischen diesen Modellen automatisch zu entdecken und die Konflikte halbautomatisch während der Entwurfsphase zu lösen. Dies reduziert deutlich die zur Konsistenzhaltung gebrachten Kosten und Bemühungen im Vergleich zu den bisherigen manuellen Lösungen.

Contents

Abstract **i**

Zusammenfassung **iii**

1 Introduction **1**

1.1 Contributions . . . 3
1.2 The structure of the thesis . . . 4

2 Foundations **5**

2.1 Model driven development . . . 5
2.2 View-based development . . . 6
2.2.1 The synthetic approach . . . 7
2.2.2 The projective approach . . . 8
2.3 Vitruvius . . . 9
2.3.1 Vitruvius development process . . . 10
2.3.2 Synchronization mechanism in Vitruvius . . . 12
2.3.3 Integration of a legacy model in Vitruvius environment . . . 13
2.4 MIR Language . . . 15
2.4.1 MIR syntax . . . 16
2.5 AMALTHEA . . . 17
2.5.1 Overview of AMALTHEA data model . . . 18
2.5.2 Model-based development process: . . . 18
2.6 ASCET . . . 19
2.7 SysML . . . 22

3 Automotive system development in the context of Bosch case study **25**

3.1 The challenges of automotive system development . . . 25
3.2 Overview about the automotive system development by Bosch . . . 26
3.3 Bosch case study . . . 26
3.3.1 AMALTHEA modeling process . . . 28
3.3.2 ASCET modeling process . . . 30
3.3.3 The integration of the ASCET and AMALTHEA models . . . 32
3.4 The consistency problem in Bosch case study . . . 32
3.5 Research question . . . 33

4 Applying Vitruvius approach in automotive systems development **35**

4.1 Research objective . . . 35

4.2 Using MIR language to declare the consistency rules 36
4.2.1 The reason of using MIR language 36
4.2.2 The evaluation mechanisms 36
4.2.3 The problems by MIR language and the suggested solutions . . . 37
4.3 Development process using Vitruvius 53
4.4 Support the scenario of legacy models 54
4.4.1 S1: Describe the correspondences between the metamodels of legacy models 57
4.4.2 S2: Build the correspondences between legacy models 58
4.4.3 S3: Find and resolve inconsistent cases 59
4.5 Modeling using external tools 59
4.5.1 Using Vitruvius as a view-based development approach 61
4.5.2 Using Vitruvius to ensure the consistency between legacy models 62

5 Evaluation **65**
5.1 Evaluation of the correspondences rules expression using MIR language . 65
5.1.1 The main correspondences between the case study metamodels . 65
5.1.2 The evaluation and the feedback of the found correspondence rules 72
5.1.3 The result of MIR evaluation 72
5.2 Applying the extended Vitruvius development process 73
5.3 Support and integration of the legacy models by Vitruvius 74
5.3.1 The mapping of the correspondences 74
5.3.2 The integration of AMALTHEA and ASCET models 75
5.3.3 Results 76
5.4 Vitruvius development process using external tools 79
5.5 Evaluate the consistency preservation by Vitruvius in practice 81

6 Related work **85**
6.1 Consistency preservation in automotive system development 85
6.1.1 Consistency preservation using document-based approach 85
6.1.2 Consistency preservation depending on model-based approach . 86
6.2 Integration of the legacy models in the change-based development 89

7 Conclusion **91**
7.1 Summary 91
7.2 Future work 92

Bibliography **93**

8 Appendix **97**
8.1 Abbreviations 98
8.2 AMALTHEA models 99
8.2.1 Software model 99
8.2.2 Components model 103
8.2.3 Runnable items 104

8.3 ASCET model 104
8.4 MIR examples 105

List of Figures

1.1 Automotive system modeling from [18] 2

2.1 View and view type terminology from [20] 6
2.2 The effort of synchronization in synthetic approach of model-based development from [14] 7
2.3 The effort of synchronization in projective approach of model-based development from [14] 8
2.4 Example of a SUM metamodel and some views for component-based software engineering from [27] 10
2.5 Use cases for developer roles in Vitruvius from [15] 11
2.6 Process for the creation of the modular SUM metamodel from [15] 12
2.7 Editing workflow in views[15] 13
2.8 Integration process for code using the linking integration strategy from [29] 14
2.9 The main steps of the model-based development using AMALTHEA [33] 19
2.10 The development process of ASCET 20
2.11 ASCET block diagram editor 21
2.12 Relationship Between SysML and UML from [18] 22
2.13 SysML diagram types from [18] 24

3.1 Layout of the control algorithm ECU 27
3.2 One of the scenarios used by Bosch for automotive system development 27
3.3 IBD diagram of the control algorithm case study 28
3.4 Automotive system development using AMALTHEA from [53] 29
3.5 The software architectures of the case study (Tasks, Runnables and labels) that are defined using AMALTHEA environment 30
3.6 The software distribution and memory mapping in AMALTHEA from [33] 31
3.7 Modeling of control algorithm in ASCET using block diagram editor . . 32

4.1 Composite pattern structure illustrates the multiple ways to refer to on object of class B. 38
4.2 Example of referring to object (**TaskRunnableCall**) using undefined order of objects (**LabelSwitch**, **ProbabilitySwitch** and **CallSequence**) . . 40
4.3 Example of mapping one-to-many relation 41
4.4 The task type in ASCET 42
4.5 The Task and ISR types in AMALTHEA 42
4.6 Examples of one way association, in each one the metaclass A is not able to access the class B directly 51
4.7 Example of no direct access problem 51
4.8 The metaclass of AscetModule 52

4.9 The updated use cases for developer roles in Vitruvius. Use cases colored with red are performed in this work. Use case colored with green will be performed in the future work. Old use cases is colored with gray. 55
4.10 Integrate structural models into change-based development approach . . 56
4.11 Resolve conflicts during the integration of legacy models. 60
4.12 Vitruvius development process using external tools. 62
4.13 Using Vitruvius to resolve the inconsistency of the models, which are created and modified using external tools 63

5.1 The component type in AMALTHEA 66
5.2 The Label type in AMALTHEA 67
5.3 ASCET data types 68
5.4 The inconsistent legacy models of the test case before the integration into Vitruvius platform (the round rectangles represent the files of the models and the bulleted list represent the model's elements). 76
5.5 The resulting consistent test case models after resolving the inconsistency by applying suggested synchronization algorithm.(the round rectangles represent model files and the bulleted list represent the model's elements. Elements created by Vitruvius to resolve the inconsistency are colored with yellow). 77
5.6 The inconsistent case study legacy models. (the round rectangles represent model files and the bulleted list represent the model's elements.) 80
5.7 Case study models after the first integration in Vitruvius platform. (the round rectangles represent model files and the bulleted list represent the model's elements. Elements created by Vitruvius to resolve the inconsistency are colored with yellow) 81
5.8 The consistent case study models after the second integration in Vitruvius platform (the round rectangles represent model files and the bulleted list represent the model's elements.) 81

8.1 Metamodel excerpt for Task, ISR and Stimulus 99
8.2 Metamodel excerpt for Stimulus and Periodic 100
8.3 Metamodel excerpt for AMALTHEA data types 101
8.4 Callgraph structure 102
8.5 Components model in AMALTHEA, which is central accessible through the ComponentsModel type 103
8.6 The runnable items in AMALTHEA 104
8.7 The AscetModule type in ASCET 105

List of Tables

5.1 Examples of the correspondences between AMALTHEA and ASCET meta-models . . . 69
5.2 Examples of the correspondences between SysML and ASCET . . . 71
5.3 Summary of MIR problems . . . 73

1 Introduction

The development of large systems or System of Systems (SoS) [42] is an inherently complex process due to the varied perspectives and viewpoints considered during the development. This complexity justifies the tendency of developers to adopt the Model-Driven Development MDD approach. According to this paradigm, all concepts and entities are represented as models with a high level of abstraction, which simplifies the design independently of the used platform and allows reusability and automatic code generation.

The developers also use the view-based software engineering approach to cope with the aforementioned complexity. In this approach the information of the system is represented using different domain-specific models and languages, which increases the level of abstraction.

The view-based development according to ISO [39] is based on one of the following approaches: The synthetic approach and the projective approach. The synthetic approach defines multiple metamodels and instantiates them as multiple views, whereas the projective approach uses a common formalism in which all viewpoints of the system can be represented.

In the automotive systems development [50] the synthetic approach is the most widely used to benefit from the different specific existing modeling tools.

Therefore, specific heterogeneous models are built by different project partners using diverse tools to describe the system from different perspectives. Among these are for example system engineering models which describe the structure of the system, software models which describe the behaviour of the software and performance models which ensure that the development meets the non-functional requirement. (see the figure 1.1).

The fragmentation and redundancy of the information overall different instances of metamodels in the synthetic approach can lead to inconsistency when for example the overlapping elements are modified only in subset of the models. The synchronization between these heterogeneous models requires a great effort compared to projective approach, because these models, which share the same semantics, are separately developed using modeling notations that belong to different metamodels.

Therefore, the project partners depend on the exchange of the information and documentation along the development process to develop consistent models. This process, however, is expensive because some of the exchanged files are written per hand. Moreover, each modification of one of the models requires new synchronization, which is also performed manually and takes more time. That is because it should be performed between each pair of the models on one hand and the mapping between the metamodels is either hidden in the modeling tools or not explicitly defined on the other hand.

Furthermore, the manual specification and synchronization could lead to inconsistency between the models developed separately. Detecting the eventual inconsistency is only possible at the assembly stage when the different previously generated codes are inte-

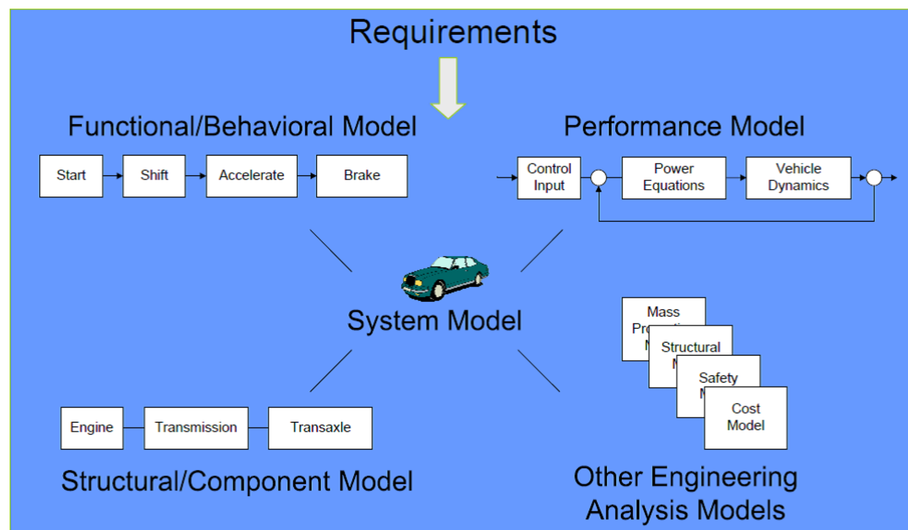

Figure 1.1: Automotive system modeling from [18]

grated and compiled. The conflicts will cause compiling or linking errors. These require much effort to be resolved as each compiling process may take hours. Besides that, the inconsistency can cause drift and erosion between the models and the implementation because keeping the models synchronized with implemented changes made to resolve the inconsistency is an error-prone and time consuming process as long as there is no proper tool that supports or verifies the synchronization between the models and their implementation.

The problems discussed above emphasize the need for an efficient mechanism to synchronize the automotive models and check the consistency between them.

The problem of synchronization between multiple views is addressed by the projective approach of view-based development, which eases the synchronization through representing the views in a single model. This approach is adopted by the OSM (Orthographic Software Modeling) [1] concept, which stores the whole information of the system in a single underlying model (SUM) so that the end users can generate their views from this SUM and synchronize them again to the SUM. This approach, however, cannot be used in the automotive industry considering that the SUM would be too general and the views will be specific to a project and cannot be reused.

The Vitruvius approach is based on the OSM concept [13]. This approach combines the different legacy metamodels of the system to serve as a modular underlying model instead of a monolithic (SUM) by OSM. The modular SUM is constructed from the legacy metamodels coupled with the consistency rules and synchronization mechanisms. The developer can consequently generate flexible views dynamically according to their roles by instantiating parts of the SUM called view types. In this way, the Vitruvius approach avoids the restriction of reusability by OSM and eases the synchronization between the views through recording atomic changes and propagating them only to the related artifacts from the SUM the model-based transformations.

For these reasons, this work is based on Vitruvius approach in order to develop and maintain consistent automotive models. This requires Vitruvius to support the further developments of existing legacy models on one hand and to allow modeling using specific modeling tools on the other hand. However, the current prototype of Vitruvius only supports the development of new views generated from the SUM and does not provide a mechanism to further develop legacy models created using other modeling tools.

1.1 Contributions

The main contribution is extending Vitruvius approach in order to support the further development of legacy models and the use of external domain-specific modeling tools during applying Vitruvius approach. The main purpose is to apply Vitruvius in order to improve automotive system development (as type of SoS), where the consistency should be preserved between large number of models (including legacy models) developed separately from different project partner using diverse modeling tools. To achieve this purpose the following sub-contributions are performed and evaluated by a case study from the AE division of company Robert Bosch GmBH [1]:

Extend the development process of Vitruvius The Vitruvius platform is already prototypically implemented. My contribution is defining and implementing the new processes which have to be included in order to support automotive system development. The new processes are importing of legacy models in Vitruvius platform, integrating them and resolving the inconsistencies if any are found.

Moreover, I will describe two scenarios to develop heterogeneous consistent models using Vitruvius. The first one is based on continuously using Vitruvius to synchronize and resolve the inconsistency despite of the modeling using external tools. The second one uses Vitruvius from time to time in order to check the consistency of the models and resolve potential conflicts.

Strategy to integrate the legacy models in change-based platform In this work I introduce a strategy to integrate different legacy models in change-based platform like Vitruvius. This strategy is evaluated already by integrating legacy automotive models of a case study from Bosch in Vitruvius platform.

Evaluation of the declaration of consistency rules using MIR language In this study I evaluate the declarative representation of consistency rules using MIR (Mapping Invariants Responses) language, which is supported by Vitruvius and defines the correspondences between the different metamodels in purpose of creating a modular SUM. The evaluation of MIR is based on using it to declare the correspondences between the metamodels of Bosch case study (AMALTHEA, ASCET and SysML). As a result I discuss in this work the

[1] see `http://www.bosch.de/de/de/our_company_1/business_sectors_and_divisions_1/automotive_electronics_1/automotive-electronics.html`

found problems and the points which have to be developed. Furthermore I suggest four solutions to improve MIR.

1.2 The structure of the thesis

The thesis is structured as follows: the next chapter 2 gives an overview about the main foundations. In this chapter I describe the Vitruvius approach in more detail. I will also introduce the main standards used by Bosch and related to the case study, which are: SysML, AMALTHEA and ASCET.

Chapter 3 describes the development process of the automotive system and explains one of the development scenarios followed by Bosch through examples of the case study.

Then the inconsistency problem encountered during development process is illustrated and the related research questions are arisen.

Chapter 4 begins with the research objectives and describes how the automotive system as an example of system of system can be developed using Vitruvius.

The evaluation with the answers of the research questions will be discussed in chapter 5. After that related work is discussed in chapter 6.

Chapter 7 gives summary and draws some conclusion.

2 Foundations

This chapter presents the conceptual and technical foundation, on which this work based. First, I will give a brief overview of model-driven and view-based development. Section 2.3 presents Vitruvius approach, which is applied in this work to keep the consistency between the automotive models that are used by the case study of the AE division of Bosch GmBH. Section 2.4 introduces the MIR language that is used to declare the semantic relations between automotive metamodels.

Then I will introduce the following models that are related to the case study of this work. Section 2.5 outlines the AMALTHEA platform. An overview of ASCET tool suite is given in section 2.6. A description of the SysML standard follows in section2.7.

2.1 Model driven development

Model Driven Development MDD is a development methodology which puts the model in the center of the attention and considers that "*Everything is a model*" [8]. The model can be defined according to Stachowiak [47] as a formal representation of natural or artificial original elements. This representation contains generally only some attributes of the original that it represents, which are selected by the model creator and relevant to achieve a certain purpose (pragmatism).

Jean Bezivin defined the MDD or as he called it Model Driven Engineering MDE as *"a set of well defined practices based on tools that use at the same time metamodeling and model transformations, to achieve some automated goals in the production, maintenance or operation of software intensive systems"* [8].

According to this definition MDD aims to utilize domain-specific languages to create models that express application structure and behaviour in a more efficient way. These languages are defined using metamodels. These metamodels are also defined using other metametamodels like Meta-Object Facility (MOF) [35], which is standardized metamodel from Object Management Group OMG (an international, open membership, not-for-profit technology standards consortium [1]). This eases the transformations between the models that are instances of different domain-specific languages. Sequentially, the MDD process will reduce the platform complexity.

MDD can be applied in three areas. The first one is the development automation for the purpose of creating new system. In this process the defined models are transformed into executable code using model transformations. The second application is the reverse engineering for the purpose of legacy modernization. The models will be extracted for more understandability of the system under study. The last application is the synchronisa-

[1] see `http://www.omg.org/`

tion using model driven transformations in order to achieve system interoperability and platform independence through combining the two former applications.

The main goal of the MDD process is increased development speed with code generation in addition to the better software quality through working and analysing models. Moreover, MDD increases the level of abstraction and simplification via models and optimizes the separation of concerns using different models (as it will explained in the next section 2.2).

2.2 View-based development

View-based development is a development paradigm that focuses on the separation of the different concerns through forming different views, each of them describes the system depending on domain-specific perspective. As a result, the whole system will be precisely described based on multiple viewpoints and level of abstraction.

There are different definition for views. In following I will present two definition that are more precise and related to my work. Atkinson et al.[1] define views as following: *a view is a normal model which just happens to have been generated dynamically for the purpose of allowing a user to see the system from a specific viewpoint.* Goldschmidt et al. [20] define also the views in addition to the main terminologies (*view type, view point*) that are shown in figure 2.1 as the following: *A view type defines the set of metaclasses whose instances a view can display. It comprises a definition of a concrete syntax plus a mapping to the abstract metamodel syntax. The actual view is an instance of a view type showing an actual set of objects and their relations using a certain representation. A view point defines a concern.*

Figure 2.1: View and view type terminology from [20]

The view-based approach is represented by ISO 42010 standard (published from International Organization for Standardization ISO in 2011 [39]) and classified into two approaches. The first approach is the synthetic approach which describes the system using different modeling notations belong to different metamodels. The second approach is the projective approach, which defines a general metamodels, which can be instantiated to represent the diverse viewpoints.

The following subsections give more details about these approaches.

2.2.1 The synthetic approach

In the synthetic approach the system can be described using multiple metamodels. These metamodels are defined and instantiated as multiple views. So, the overall system will be synthesis of the information resulted of these views. The advantage of this approach is the distribution of the complex system description on different metamodels addressing different aspects of the system.

However, the information of system, which can share the same semantics, will be distributed across different views in this approach. The fragmentation of the information can lead to inconsistency, either because of the redundancy of a piece of information or the modification of the overlapping elements only in some views. As a result, the synchronization between each pair of views will be needed. The number of these synchronizations will grow as a result quadratically with the number of the views (see figure 2.2). Furthermore, the synchronization will be more difficult and need more effort if they are performed manually.

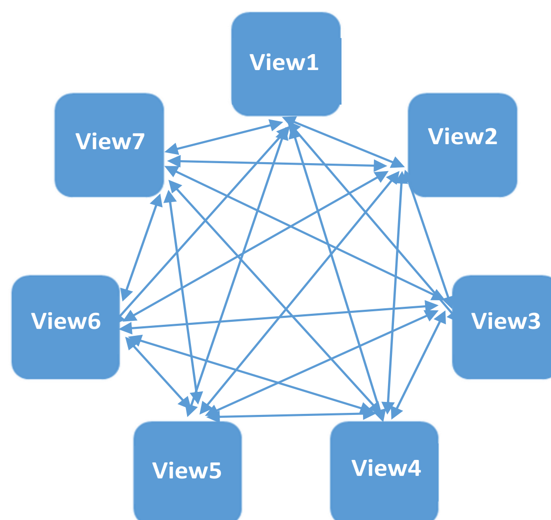

Figure 2.2: The effort of synchronization in synthetic approach of model-based development from [14]

2.2.2 The projective approach

This approach uses a common formalism, in which all viewpoints of the system can be represented. In other words each view can be generated from a single base metamodel by hiding details that are not relevant for the particular viewpoint taken into account.

Contrast to syntactic approach the synchronization between the views will be easier, because of representing the all views in a single model. As a result, each view has to synchronize itself only with the central model and the information will be propagated from the central model to the other views (see figure 2.3). So, the synchronization relations grows only linearly with the number of views.

The projective approach suffers that the central metamodel has to represent the all viewpoints and be compatible to the existed formalisms. Moreover, this central metamodel will be suitable only for specific development scenarios. In other words, the reusability is low.

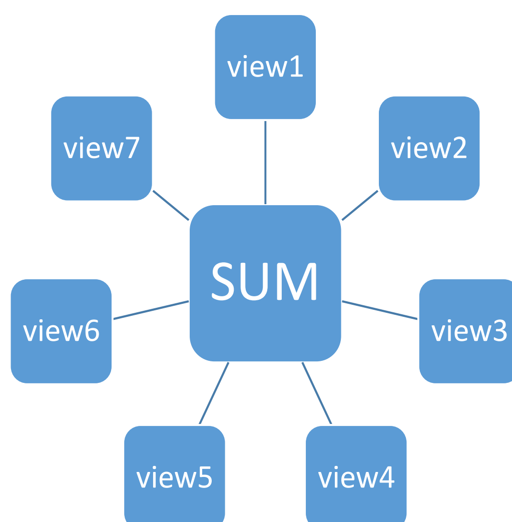

Figure 2.3: The effort of synchronization in projective approach of model-based development from [14]

This approach is adopted by OSM (Orthographic Software Modeling) [1] concept, which stores the all information of system in a single underlying model (SUM). So, the information will be available to all developer, who can generate their views automatically from this SUM metamodel, modify them and synchronize them with SUM.

The automatically generation of views according OSM approach will be based on model transformation from the SUM. Similarly, model transformation are used for keeping the views consistent, since the changes in a view is reflected in SUM, which synchronize them with the other all other views. Hence, bi-directional transformation for every view is needed. The complexity of writing these transformation in OSM architecture is linear and less significantly than the complexity of writing them in peer-to-peer architecture like the case of synthetic approach (see the different between figure 2.2 and figure 2.3)

The OSM approach considers however that the SUM will be too general to represent all information. Besides that,OSM does not provide a way for constructing the SUM. In the

common component modeling example (CoCoME) [2] Atkinson et al. built the metamodel for SUM manually in order to apply OSM concept.

2.3 Vitruvius

Vitruvius (VIew-cenTRic engineering Using a VIrtual Underlying Single model) approach is view-based software development approach. It enables express the system from various viewpoints based on OSM concept [1]. Contrast to OSM approach Vitruvius approach combines various heterogeneous metamodels in addition to the information about the correspondences between them to serve as a modular SUM [14].

As a result, Vitruvius approach combines the advantages of the projective and synthetic approach. It enables the usage of the different domain-specific metamodels and presents concept to ease the synchronisation and to avoid the inconsistency and complexity.

According to Vitruvius approach the developers can also generate flexible views [14] with more details about their roles in addition to the combined abstract view about the system. These views integrate information from instances of multiple metamodels, enable the selection of related elements, identifies the overlapping elements by a naming convention and sets the editability restriction.

For example Vitruvius concept has been applied for component-based development scenario [14, 13, 15, 27]. This scenario is based on three formalisms: the Palladio Component Model (PCM) [6] for representing both of software architecture and performance properties, Unified Modeling Language (UML) class diagram for representing class architecture and Java code for the implementation. Figure 2.4 represents the SUM metamodel of this example as a circle containing several view types that are parts of the legacy metamodels of PCM, UML and Java. Developer of system can instantiate views from these defined view types according to their roles.

For example system architects instantiate component diagram view from view type VT3, component developers instantiate UML class diagram from view type VT1 and programmers instantiate Java source view from view type VT4. Moreover, the multi-scope view type VT2 enable that system architects and component developers instantiate a view with information from both PCM component and UML classes metamodels in addition to information on which classes implement which component. The resulting view (component-class implementation view) is an example of the flexible views. It represents elements from two distinct metamodels in addition to the relation between them, displays only components and classes that are connected by an implementation relation and allows editing only the implementation relation (both components and classes are read-only).

SUM metamodel contains also the semantic relations and consistency constraints between the legacy metamodels, which are used to guarantee that the SUM is always in consistent state. These relations and constraints are defined through determining Mapping-Invariants-Response (MIR) elements: the mapping between the metamodels elements, the invariants (consistency constraints) and the responses that applied to preserve the consistency when the constraints are violated. For this purpose Mapping-Invariants-Response (MIR) language can be used as it will be explained in the section 2.4. These relation and

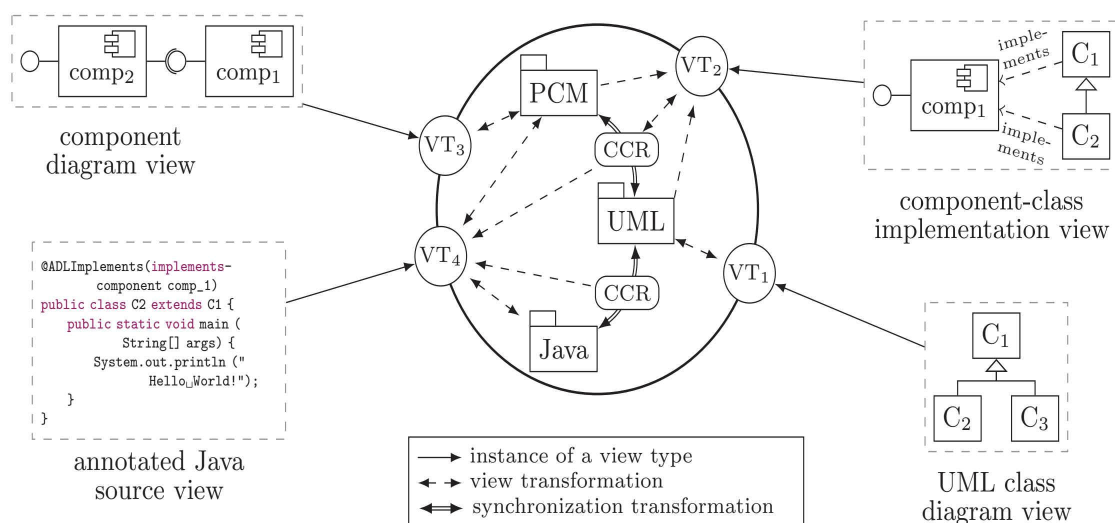

Figure 2.4: Example of a SUM metamodel and some views for component-based software engineering from [27]

constraints are defined only where there is semantic overlap like the case between PCM metamodel and UML metamodel.

In addition to the abovementioned case study Vitruvius approach has already tested with case studies based on PCM, Java Modeling Language (JML), and Java source code [28] [29].

2.3.1 Vitruvius development process

Vitruvius development process makes a difference according to the role of users (as it is shown in figure 2.5). The main users are the methodologist and the developer. In the following, I will explain the development process of Vitruvius for each user role.

Development process according to the methodologists Vitruvius development process from the viewpoint of the methodologist (as shown in figure 2.5) can be summarized into the following points:

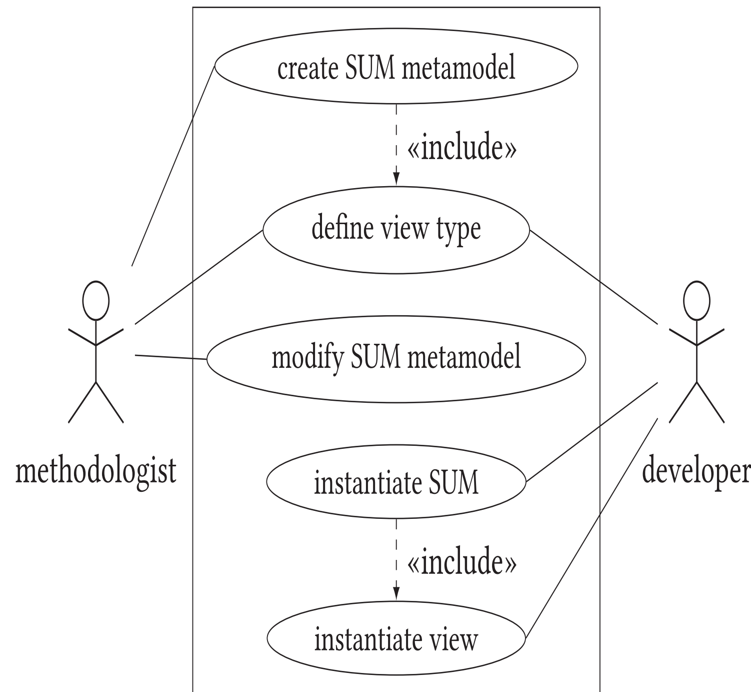

Figure 2.5: Use cases for developer roles in Vitruvius from [15]

- Creating VSUM metamodel through adding the legacy metamodels, defining at least a view type for each metamodel and determining the correspondences between the metamodels. (SysML activity diagram in figure 2.6 illustrates the process of creating modular SUM metamodel). The definition of the correspondences is done by describing the MIR elements (Mapping, Invariants and Responses) between each pair of metamodels, where there is semantic overlap. MIR elements are the mapping between the metamodels, the invariants, which determine the main constraints, and the responses, which describe the suitable action, which have to be performed when the invariants are violated. For this goal, the declarative language MIR (2.4) or another transformation language can be used (like Xtend language [2] or QVT language [36]).

- Methodologists can elicit a set of the metamodels to define and add combined view types to the SUM. This requires also, that the correspondences between the selected metamodels are defined by the last step.

The development process according to the developers The developers use the predefined view types, that are created from the methodologists to access, generate and manipulate

[2] see `http://www.eclipse.org/xtend/`

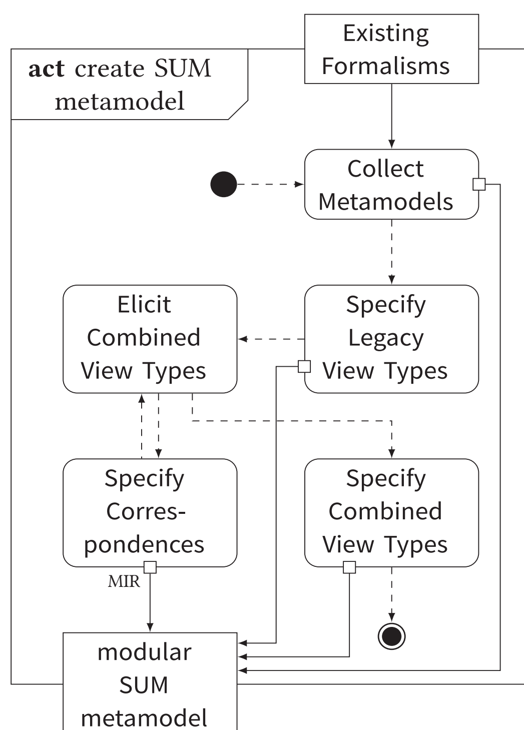

Figure 2.6: Process for the creation of the modular SUM metamodel from [15]

their own views. Moreover, they are allowed to define their custom view types (see figure 2.5).

Vitruvius will offer two types of collaborative development. The first type is online-synchronous modeling which allows the developer to check out their views, modify them and synchronize them online. The developers in this type have to be connected to a central repository. As a result, the occurred changes can be propagated immediately and the developers will be notified about them. To prevent the conflicts a versioning system will be supported, which creates a new version after each editing process to the parts, that are edited synchronously.

The second type is the asynchronous modeling. The developers using this type will be able to check out their working copies, modify them, and check in them again. However resolving the concurrent conflicts will be required by each check-in.

2.3.2 Synchronization mechanism in Vitruvius

Vitruvius depends on the change-based approach to preserve the consistency between the generated views. According to this approach, the view is marked as dirty after any modification and all atomic changes made to the view are recorded. The user has to save its view in order to reflect these changes to the VSUM and make them also available by the next re-opening of the views. When the developer saves the view, the changes are synchronized with the related parts of SUM. For this purpose, each change triggers an appropriate model-based transformation according to the type of modification (create,

edit or delete) and the modified artifact. These change-driven model transformations will propagate the changes to the related artifacts from VSUM. Potential conflicts can appear after applying the transformation because of violating sub-metamodel constraints, inter-metamodel constraints defined by MIR elements, or both of them. To resolve the conflicts the responses actions defined by MIR elements are applied automatically. These responses are defined by methodologists for every invariants. Applying the responses can also lead to further responses. Moreover, not all conflicts may be resolved automatically. In this case, manual operations may be needed to preserve the consistency, otherwise the changes will not applied to the modular SUM and the state of the view will not change (the state stays dirty). (SysML activity diagram showed in figure 2.7 illustrates the abovementioned view editing steps)

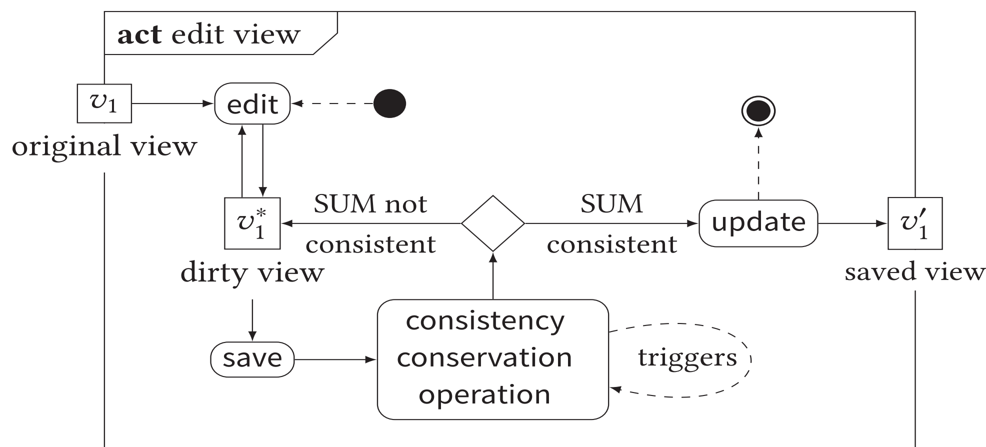

Figure 2.7: Editing workflow in views[15]

MIR language aims to generate the change-driven transformation automatically from the declarative expression of the correspondences rules between the metamodels [27, 28].

2.3.3 Integration of a legacy model in Vitruvius environment

Vitruvius approach support the development of complex systems, which consist of heterogeneous models. These models, which often share the same semantics, are kept consistent based on change-based approach. To benefit from Vitruvius approach for developing consistent models, it will be also useful to integrate the already existing legacy models into Vitruvius environment. These legacy models are often developed using external modeling tools, which are not supported from Vitruvius.

The integration of these legacy models in Vitruvius environment will not be possible without information about all the changes that have been made to these models. In other

words, the developers, who want to integrate legacy models or to use external tools for modeling, have to record all the atomic editing steps, detect the occurred changes through calculating the differences between two versions of the model or to find a way to generate the occurred changes when they are not available. In the following subsection I will talk about two strategies to integrate only one legacy model into Vitruvius environment.

The strategies of integrating a legacy model There are two strategies to integrate a legacy model into Vitruvius platform. The first one is Reconstructive Integration Strategy (RIS) [29]. In this strategy the change histories of the legacy models have to be generated, as if this model is recreated. Then these changes are transmitted to Vitruvius which propagates them in SUM depending on incremental change-driven model transformations. As a result, the legacy model is integrated in Vitruvius and the applied transformations create its corresponding models. This strategy has been validated with case study, which integrates the Palladio Component Model PCM through detecting and recreating the changes and then using them to create the linked elements (Java code elements).

The second strategy is Linking Integration Strategy (LIS) [29]. It creates the correspondences between the legacy model on one hand and its corresponding model, which is generated using reverse or forward engineering process, on the other hand. Besides the legacy model and its generated corresponding one, LSI expects as input the set of linking information, which is created during the process of the reverse/ forward engineering. In this process the software architecture can be generated from the code and visa versa. As a result, the LSI strategy will link the code and software architecture based on the linking information. The linking information will be used to generate the correspondence model, which specifies which elements from the code elements are corresponding with which elements from the architecture model. The figure 2.8 shows the process of code integration in change-based environment using LIS.

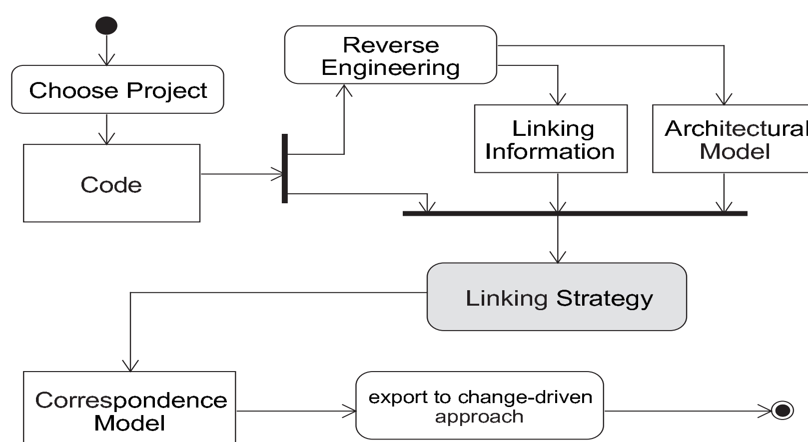

Figure 2.8: Integration process for code using the linking integration strategy from [29]

This strategy is implemented in order to add the correspondences between the elements of the java code on one hand and PCM information generated by the reverse engineering tool SoMoX (SOftware MOdel eXtractor [3]) on the other hand, depending on the tracing information provided from SoMoX (see figure 2.8).

Both the abovementioned strategies are applied only to one homogeneous legacy model (either the model as in RIS case studies or source code as in LIS case studies) and generate its corresponding model. In other words, the current version of Vitruvius does not support the integration of multiple heterogeneous legacy models in its environment.

2.4 MIR Language

Kramer et. al. introduce MIR language based on textual domain-specific language (DSL) for the purpose of express the correspondences rules between the metamodels. The correspondence rules are declarative expression of semantic relation as well as consistency constrains[48]. These rules can be used to keep the Virtual SUM(VSUM)[4] consistent [27].

MIR is defined with XText language (Framework for development of programming languages and domain specific languages) [5] in order to obtain the full infrastructure including parser, linker, type checker, compiler and editing support for Eclipse [7].

Other benefit from using Xtext is that Xtext supports using Xbase language [6]. Xbase is an expression language that can be embedded into Xtext language and offers additional advantages like type inference, lambda expressions, a powerful switch expression and a lot more.

MIR language provides the declarative expression of the corresponding rules between the metamodels through defining the following three elements:

- The mapping between the different metaclasses and features of the related metamodels.

- The invariants which define the consistency constraints between the different metamodels.

- The response actions, which have to be applied in order to preserve the consistency of VSUM, when consistency constraints are violated.

Depending on this declarative description of the correspondence rules MIR language aims to generate the synchronisation model-based transformations between the metamodels automatically, which are required to preserve the consistency between the models.

[3] see `www.somox.org` and `http://www.q-impress.eu`

[4] VSUM refers to the term *modular SUM*, which is used by Burger in [14],[13],[15]. From this point VSUM term will be used instead of modular SUM

[5] see `http://www.eclipse.org/Xtext/`

[6] see `https://wiki.eclipse.org/Xbase`

2.4.1 MIR syntax

Current MIR editor allow only writing the declarative mapping between two metamodels. The invariants and responses are not implemented.

MIR describes the correspondence rules textual in (.mir) file, which is based internally on XText language. The MIR file begins with importing the required metamodels and the required bundles containing the model codes of the metamodels. Then various map blocks can be declared between the metaclasses of the imported metamodels.

The structure of the MIR file is shown in listing 1.

```
generates package vitruv.test1
generates type TestMIRExecutor

bundle org.first.firstMetaModel
bundle org.second.secondMetaModel

import package "http://www.first.org/firstMetaModel" as mm1
import package "http://www.second.org/secondMetaModel" as mm2

map mm1.MetaclassA as a and mm2.MetaclassB as b
{
  when-where { ... }
  with-block { ... }
  with map ... and ... { ... }
  ...
}
map mm1.MetaclassC as c and mm2.MetaclassD as d
{
  when-where { ... }
  with-block { ... }
  with map ... and ... { ... }
  ...
}
...
```

Listing 1: MIR file structure

Mapping block structure: Each map block defines the correspondence between two metaclasses in a bidirectional, declarative way. For this purpose the *map* key word is used. Then the correspondent metaclasses are determined. The map block can define also the following information:

- The pre- and post-conditions. The pre-condition defines the constraints, which have to be fulfilled in order to apply the mapping between two instances of the specified metaclasses. The post-conditions are enforced once the pre-conditions hold. The **when-where** block is used to express these conditions.

For example the **when-where** condition defined in listing 2 is checked from left to right (*when*) as literal equality checks (metaClassVarName.attributeName == Literal-Expr) and is enforced from right to left (*where*) as literal assignment expressions (metaClassVarName.attributeName = LiteralExpr).

```
generates package vitruv.test1
generates type TestMIRExecutor
bundle org.first.firstMetaModel
bundle org.second.secondMetaModel
import package "http://www.first.org/firstMetaModel" as mm1
import package "http://www.second.org/secondMetaModel" as mm2

map mm1.MetaclassA as a and mm2.MetaclassB as b
 {
        when-where
        {
              equals(a.attributeName,LiteralExpr)
        }
 }
```

Listing 2: Example of when-where expression

- The mapping of the attributes, which need a certain computation using restricted conversion operators, is defined using **with-blocks**.

- The nested mapping which declares the correspondence between the attributes and the references. For this purpose the **with map** blocks are used.

2.5 AMALTHEA

AMALTHEA [7] [33] is an open and expandable tool platform for embedded multicore systems. It is developed in a publicly funded ITEA 2 project [8] [32] , to combine the different tools used to develop multi-core automotive ECUs (Electronic Control Unit) in a single platform, which eases the exchange of data between them [10]. AMALTHEA combines furthermore information of the low level behavior and timing, which enables the simulation and verification of the systems.

AMALTHEA architecture is compatible with the standardized AUTomotive Open System ARchitecture, AUTOSAR [3]. That means it can support the multicore automotive systems well, which increases the performance of processing the growing numbers of complex functions needed to provide the increasing comfort and driver assistance systems.

[7] see `http://www.amalthea-project.org/`

[8] see project information in `https://itea3.org/project/amalthea.html`

2.5.1 Overview of AMALTHEA data model

AMALTHEA supports model-based development based on two main models. The first one is the system model, which offers the processing of data and build the models of the system. The second main model of AMALTHEA is the trace model, which analyses the results of execution/simulation of the defined system model and verifies the timing behavior.

2.5.2 Model-based development process:

Using the system model and trace model the developers follow and process the data in AMALTHEA between the following activities, which are shown in figure 2.9:

- **Modeling** In this step the behavior of the system is defined in form of block diagram or state diagram, whereas the dynamical behavior is implemented using external modeling tools like ASCET. The used hardware will also be modeled in this step. That means all the information about the hardware, like the processor, cores, memories and timers, will be described. Moreover the system constraints have to be cleared.

- **Partitioning**: In this step the defined software units will be broken into smaller parts, which can be allocated to available scheduler of hardware elements, i.e. the cores in the case of a multicore processor. The initial tasks will be identified in this step. The units that can be run in parallel (runnable) will be also determined and assigned to the tasks or Interrupt Service Routine (ISR).

- **Mapping:** Based on the software and hardware models defined in the various steps, the executable software units will be assigned to the cores of system. Furthermore, the data and the instructions will be mapped to the memory sections of hardware.

- **C code generation:** In this step the C code (glue-code) is automatically generated and the software is executed in order to evaluate the defined models and timing behaviour.

- **Tracing:** The timing traces resulted by the evaluation and saved in the trace model will be further traced and analysed to verify the correct behavior of the software system.

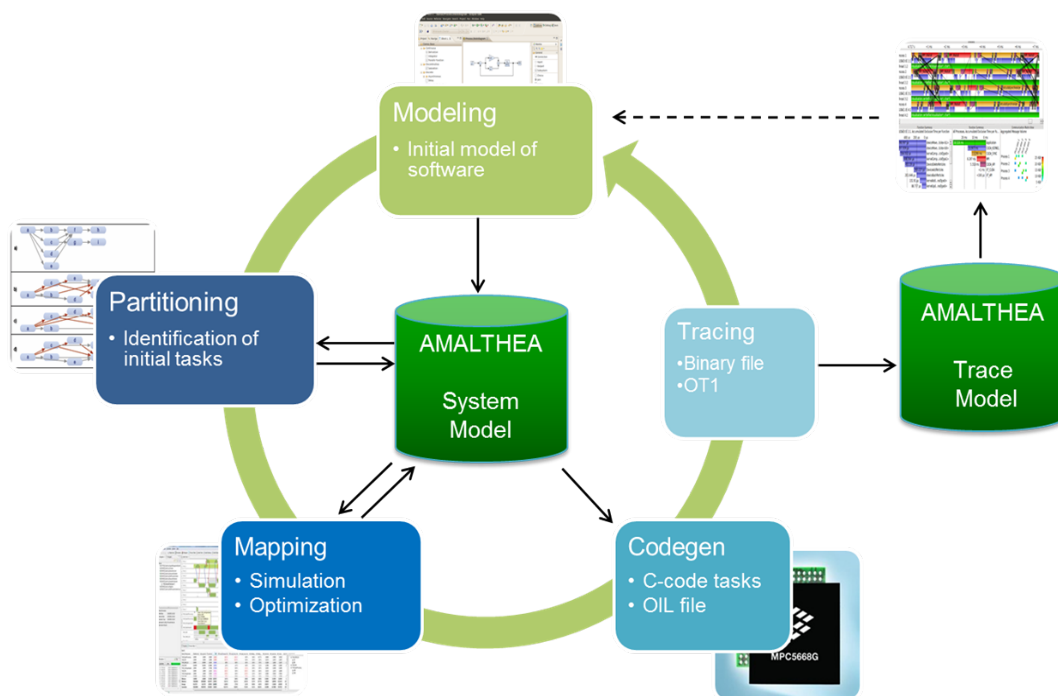

Figure 2.9: The main steps of the model-based development using AMALTHEA [33]

2.6 ASCET

Advanced Simulation and Control Engineering Tool ASCET (previously ASCET-SD) [22] [21] is a tool suite from ETAS GmbH for model-based development of embedded automotive software[23] [9].

ASCET describes and models the functionality of the ECUs using a database of components, models and data. The modeled functions are compatible with AUTOSAR standard. ASCET generates optimized C code comparing to C code written by hand. This code is also compatible with the specified ECU operating systems.

ASCET enables simulation to validate the modeled functions. In addition to the simulation ASCET offers rapid prototyping considering the real time behaviour, which can be done in labor or in vehicle. Moreover, ASCET offers automatic documentation of the modeled ECU software (figure 2.10 shows the main functionality of ASCET tool suit). ASCET consists of the following main products :

[9] see `http://www.etas.com`

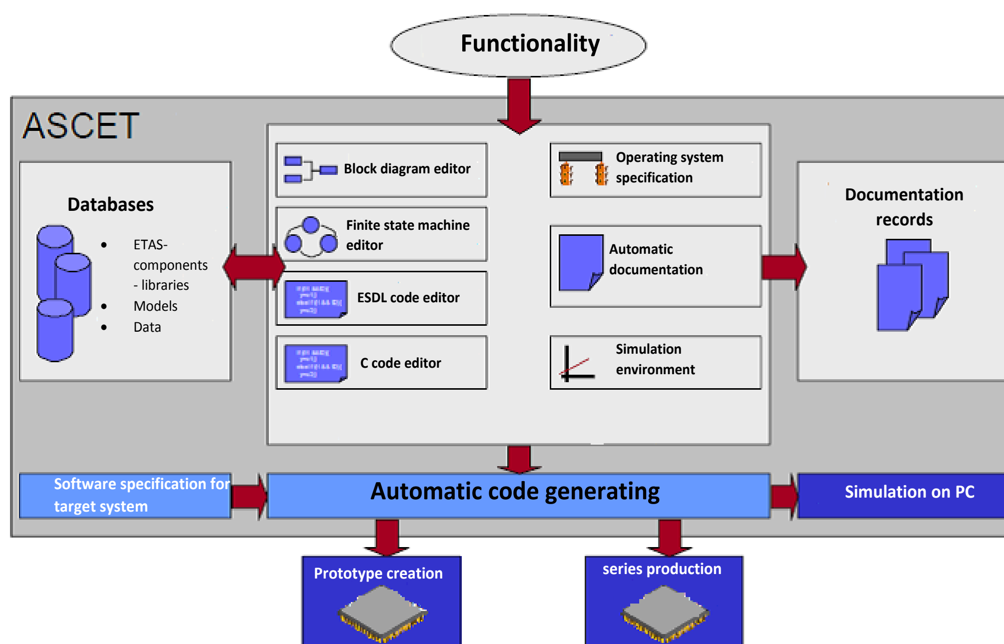

Figure 2.10: The development process of ASCET

- ASCET Modeling and Design (ASCET-MD) [10] offers executable specification of the functions.

  ASCET-MD is compatible with AUTOSAR standard [3] and supports as a result the development and the integration of its components. Not only AUTOSAR components but also the components of Matlab/Simulink [11] can be easily imported and integrated using ASCET MATLAB Integration Package (ASCET-MIP). This package is compatible with ASCET-MD and ASCET Rapid Prototyping and allows the model transformations from and to Matlab/Simulink.

  The behavior of ECUs (like real-time behavior) can be described and modeled by either graphical modeling tools like block diagram and state machine or by textual tools like Embedded Software Description Language (ESDL) editors and C code editors. In the following I will give an overview about these modeling editor:

  – The block diagram editor: It describes the functionality of the main components through blocks. Each block represents a component. The flow of the data or control signal between the component is represented through directed arrows (the solid arrows represent data flow whereas the dashed arrows represent the control flow). Different data type (parameters, messages, variables, constants etc.) in addition to the arithmetical and logical operations can be represented using different pre-defined blocks. Figure 2.11 shows an example of defining an

[10] see http://www.etas.com/de/products/ascet_md_modeling_design-details.php
[11] see http://www.mathworks.com/

ASCET component using the Block diagram editor. This example describes the functionality of the timer component using three methods. The first method *"start"* gives the signal to start. The second method *"out"* returns the out value while the timeCounter variable is greater than 0. The third method *"compute"* calculates the new value of timeCounter.

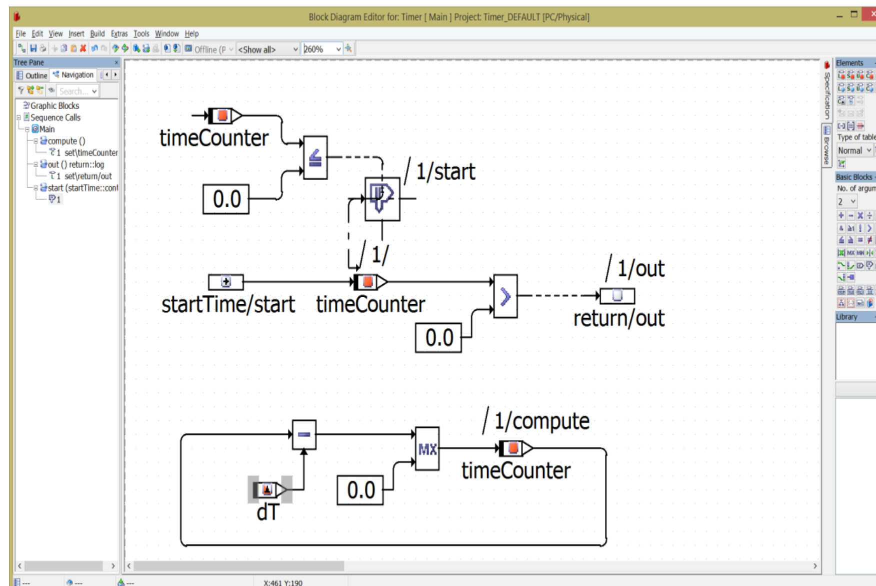

Figure 2.11: ASCET block diagram editor

- The advanced finite state machine editor: It models the different states of the system and how the different events change the current state of system.
- ESDL (Embedded Systems Description Language) editor: This language is used for modeling the physical layer of the system. It is also similar to the Java language.
- C code editor: it is particularly used for describing hardware functionality, accessing hardware drivers etc.

• ASCET Software Engineering (ASCET-SE) [12] is a tool to generate target-specific C code for selected microcontrollers in relatively shorter time and higher quality compared to handwritten code. ASCET-SE supports code generation for both OSEK Operating Systems (OSEK OS) and AUTOSAR Run-Time Environments (AUTOSAR RTE).

• RTA-OSEK [13] is ETAS' OSEK-compatible Real-Time Operating System, which can be used for all types of automotive ECUs development. It models the scheduling of the functions and as a result it describes the runtime behavior.

[12] see `http://www.etas.com/de/products/ascet_se_software_engineering.php`
[13] see `http://www.etas.com/de/products/rta_osek.php`

- ASCET Rapid Prototyping (ASCET-RP)[14] enables create a rapid prototyping of the ASCET functions in order to validate them earlier. These functions can be connected to the both of I/O and system buses and executed in order to support Hardware-In-the Loop (HIL) simulation which test the complex real-time embedded systems.

2.7 SysML

Systems Modelling Language SysML is a graphical modeling language developed for systems engineering by OMG, the International Council on Systems Engineering INCOSE (a not-for-profit membership organization founded to develop and disseminate the interdisciplinary principles and practices that enable the realization of successful systems [15]) , and a standard for the exchange of system engineering data AP233, which is a part of iso10303 standard [24] workgroup. It extends the UML as it is shown in figure 2.12 to model a wide range of systems, which may include hardware, software, information, processes, personnel, facilities and procedures [18].

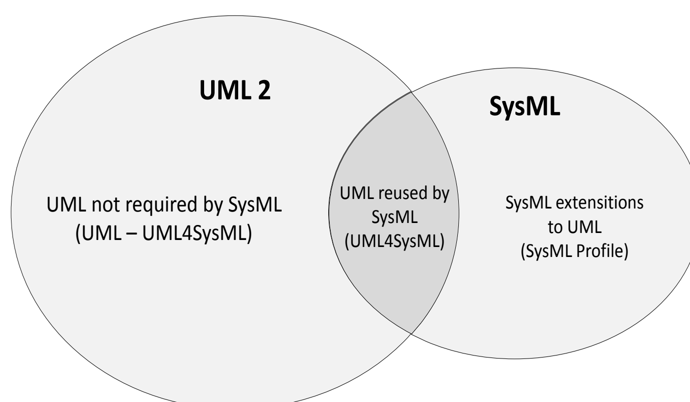

Figure 2.12: Relationship Between SysML and UML from [18]

SysML is based on model-driven approach. It separates the concerns in different architectural models/views in order to support the specification, analysis, and design of the complex systems. Furthermore SysML improves the quality of the design by detecting the errors earlier. SysML can enable the automatic code generation through some supported tools.

The four pillars of SysML The diagrams provided from SysML can describe the system from different viewpoints:

- **Structure:** The SysML can depict system structure, like the system hierarchies, system block, and the interconnections between the different parts of system. For

[14]see `http://www.etas.com/en/products/ascet_rp_rapid_prototyping.php`
[15]see `http://www.incose.org/`

this purpose the Block Definition Diagram (BDD) or Internal Block Diagram (IBD) can be used.

BDD provides a black box representation of a system block, which can be of any type including software, hardware, etc. Moreover it defines the flow ports, which represent what can go through a block (data, matter, or energy). The flow ports can be in-ports and/or out-ports.

IBD instantiate the BDD to represent the final assembly of all blocks within the main system block and depicts the interconnection between them.

- **Behaviour:** SysML provides diagrams to describe both of function-based behaviours and state-based behaviours, like activity diagram, sequence diagram, state machine diagram and use case diagram.

- **Properties:** The performance and quantitative constraints are defined and analysed using the parametric diagram (or it can be called constraints diagram), for instance in the automotive system the maximum acceleration, minimum curb weight, and total air conditioning capacity parameters can be represented and quantitative analysed using this diagram.

- **Requirements:** SysML supports the representation of the functional and performance requirements. Moreover it enables the capturing, deriving and traceability of the represented requirements. For this objective, the new added diagram called requirement diagram is used.

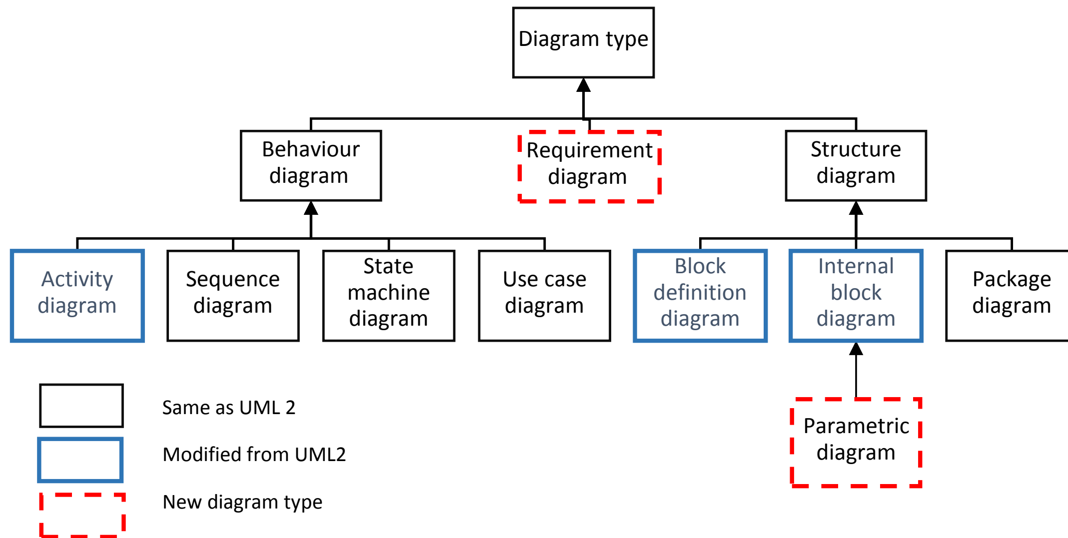

Figure 2.13: SysML diagram types from [18]

3 Automotive system development in the context of Bosch case study

In this chapter I will explain first the main challenges which the automotive system development faces in section 3.1. Then section 3.2 gives an overview about the development process followed by AE division of Bosch GmbH. In this overview I will describe some tools used by AE division of Bosch to overcome the main challenges. Then I will explain the case study and one of the development process scenarios followed by Bosch in section 3.3. In this scenario the use of automotive modeling tools will be illustrated. Moreover, the consistency problem and its reason will be explained through the case study. Based on the consistency problem I will finally introduce the research questions in section 3.5.

3.1 The challenges of automotive system development

Automotive systems became more complex in the recent years. For example the modern cars today contain more than 100 control units [38]. These units provide several services, for example operating the engine or controlling the heating, ventilation and air condition systems. Therefore several project partners and suppliers need to share and combine their experts to develop one project separately using different tools. Furthermore, the automotive system tends to use the multi-core processor in order to increase their performance on one hand and to reduce the number of the technical and economic problems of using a large number of single processors on the other hand, for instance the problems of the isolation and the limited composability of the processors [26]. Moreover, the integration of functional modules developed from multiple suppliers on the different types of ECUs became a necessity, which should be met. That requires supporting AUTOSAR standard during the development, to ensure the decoupling of the supporting hardware and software services. Multiple functions of the automotive system are based on the real time environment and consequently a suitable environment to define and validate them will be needed. Finally most of automotive services require a high level of the reliability and safety, therefore they have to be validated well and also simulated to detect defects in the earlier stages.

To overcome the abovementioned challenges the developers of automotive systems tend to use different heterogeneous domain-specific formalisms, which requires keeping these formalisms consistent along the development process as my work aims. The following section gives an overview about Bosch GmbH and some of the formalisms that it uses to overcome these challenges.

3.2 Overview about the automotive system development by Bosch

Bosch as one of the largest automotive suppliers consists of a lot of departments and teams working together with a variety of technologies to supply precision automotive components and systems. Bosch has significantly participated in the development of automotive systems using various standards in order to meet the requirements mentioned in section 3.1. For example Bosch founded ETAS GmbH in 1994 [23] which provides innovative solutions for the development of embedded systems for the automotive industry and also developed its main product ASCET. Furthermore, Bosch has managed the ITEA 2 project which develops AMALTHEA platform to support multicore systems and tool chain. Bosch uses ASCET, AMALTHEA, SysML and other different standards for automotive system development. In the following I will emphasize the purpose of using the standards related to my case study (SysML, AMALTHEA and ASCET).

- At first Bosch uses SysML to define the system context, via establishing the system boundary and all system actors (humans and external systems) that interact with the system. For this purpose package diagram, use case diagram and block diagram can be used. Moreover, it is used to describe the structure of the system including block hierarchy and the relationships between the system parts using BDD and IBD diagrams. Furthermore, both requirements and system parameters can be described and analysed by SysML.

- Secondly Bosch uses AMALTHEA to support the development of multi-core embedded systems, which are used widely in the recent years. For this purpose, AMALTHEA is used to describe the hardware architecture, software architecture and the software distribution on hardware architecture. Besides that, AMALTHEA provides simulation and timing analyses. Moreover, Bosch benefits from AMALTHEA tools chain by easing the exchange of data and results between the different project partners [53]

- The third used standard is ASCET. It is used to model, specify and implement (in term of code generation) the behavior of the software functions. ASCET supports real-time behavior and enables the definition of its execution sequence.

3.3 Bosch case study

This case study describes the development of controller software dependent on a control algorithm. This algorithm is used by the systems, whose state depends on the value read by a sensor, and which use an actuator to influence this state.

For example the control algorithm read data about the actual position of the vehicle from sensors and calculate control values for the new position which are sent to an actuator.

To achieve this purpose, the control algorithm depends on control loop feedback mechanism. It calculates an error value as the difference between a desired set point (target

position) and a measured process variable. Then it attempts to minimize the error over time by adjustment of a control variable.

The following figure 3.1 shows the layout of the final ECU, where the control algorithm will be later executed.

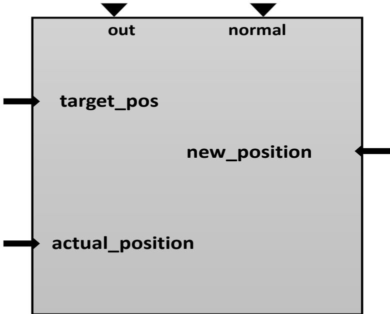

Figure 3.1: Layout of the control algorithm ECU

There are different scenarios of the automotive system development, which are used by Bosch and can be applied to develop this case study. For example, either the top-down design, bottom-up design or mixture of them can be applied. In this case study I will outline one of these scenarios, which depends on top-down design and is shown in figure 3.2.

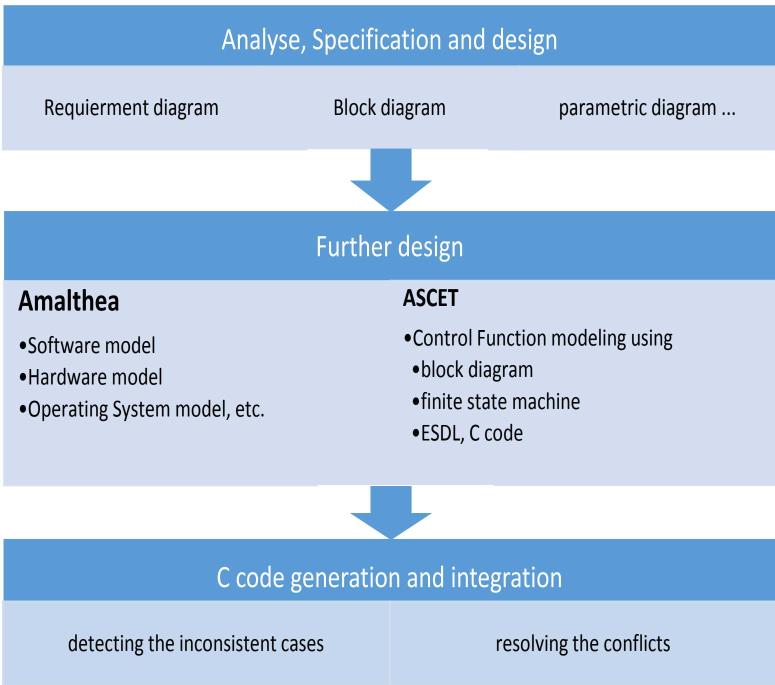

Figure 3.2: One of the scenarios used by Bosch for automotive system development

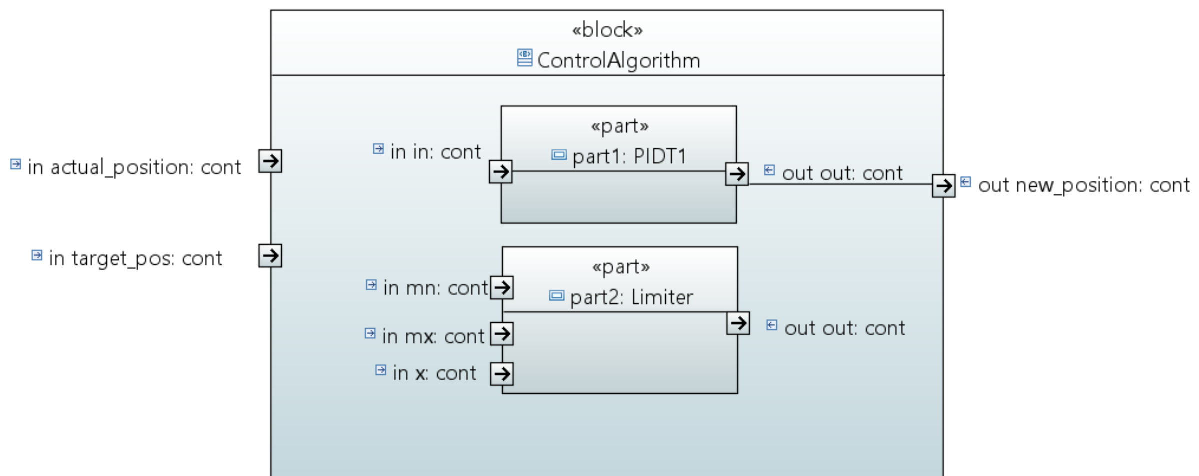

Figure 3.3: IBD diagram of the control algorithm case study

In this scenario the automotive system is analysed and specified (this step can be done by SysML. Actually SysML is not always used in this step but Bosch AE aims to use it in future). Then the system architects share the resulting information with the other project partner in order to build and maintain the AMALTHEA and ASCET models separately. The resulting codes of these models have to be integrated with each other.

The structure of the controller can be defined by SysML using the BDD and IBD diagrams.

The *ControlAlgorithm* software block is responsible for applying the control algorithm. Therefore it receives the information about the actual and target position through its in-ports and sends the result by `new_position` out-port. The figure 3.3 shows the IBD diagram of the controller system. This diagram represents the main Block (ControlAlgorithm) with its internal blocks (PIDT and Limiter) in addition to the in-/ out-ports (actual_position, target_position and new_position).

In the following I will explain the modeling process of AMALTHEA that is needed to define software architecture of the ControlAlgorithm block as well as the modeling process of ASCET that is required to describe the functionality of this block. After that I will explain the problem of integrating the code generated from these models.

3.3.1 AMALTHEA modeling process

The software developers of AMALTHEA prefer to use agile development process because of the complexity of automotive systems [4]. Therefore, the development process will be broken down in fast iterations. In each iteration the requirements of this iteration will be defined and different variants of the final system will be described too.

According to the requirement and the selected variants the modeling, partitioning, mapping, code generation and tracing activities (mentioned in section 2.5.2) will be applied step by step (see figure 3.4). After the requirement engineering and variants modeling, the software architecture will be defined using components to describe the structure of

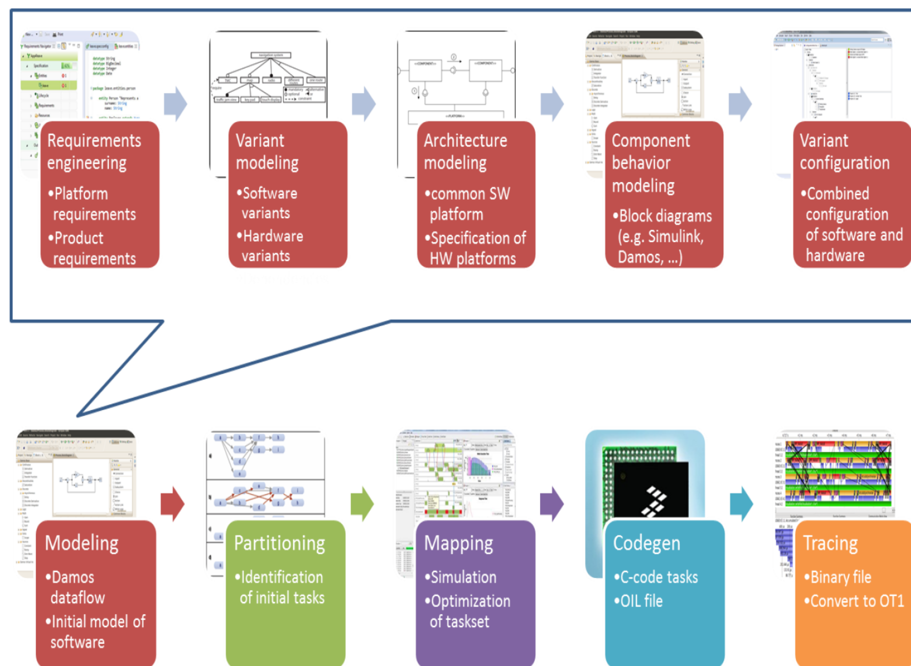

Figure 3.4: Automotive system development using AMALTHEA from [53]

the automotive system. However, these components will be implemented and simulated using other tools. In this case study Bosch uses ASCET to implement the defined software components. For example, AMALTHEA defines a component called ControlAlgorithm component (its functionality will be illustrated in the next section 3.3.2). The defined software architecture will be saved in the components model.

Besides that, the hardware model of the system is described in modeling step. This model includes detailed and accurate information about the hardware configuration, like data about the number of cores and their frequency, the type of memory (shared memory, distributed memory, private memory), the operating system, the scheduling, the timing constraints, process communication, etc. Moreover, the constraints of the hardware are defined in this step too.

In partitioning step, the developers define the software model elements that can be allocated to the hardware elements. Here, I will explain the software elements, which are belong to the case study. The developer determine the smallest executable parts of code, which can be run in parallel. These parts are called runnables. The initial activation of a runnable can only be performed by a task or another runnable. Therefore, the developers define the tasks and add them to the software model. Tasks provide context for the operating system. For instance, in this case study a task called Task_10MS is defined in the software model. This task calls two runnables (ControlAlgorithm_normal and ControlAlgorithm_out). The following figure 3.5 shows the task of the ControlAlgorithm component and how it calls the runnables.

Moreover the software model defines three labels. The labels represents the data elements and will be directly located in a defined area of a given memory. These labels are accessed from the defined runnables.

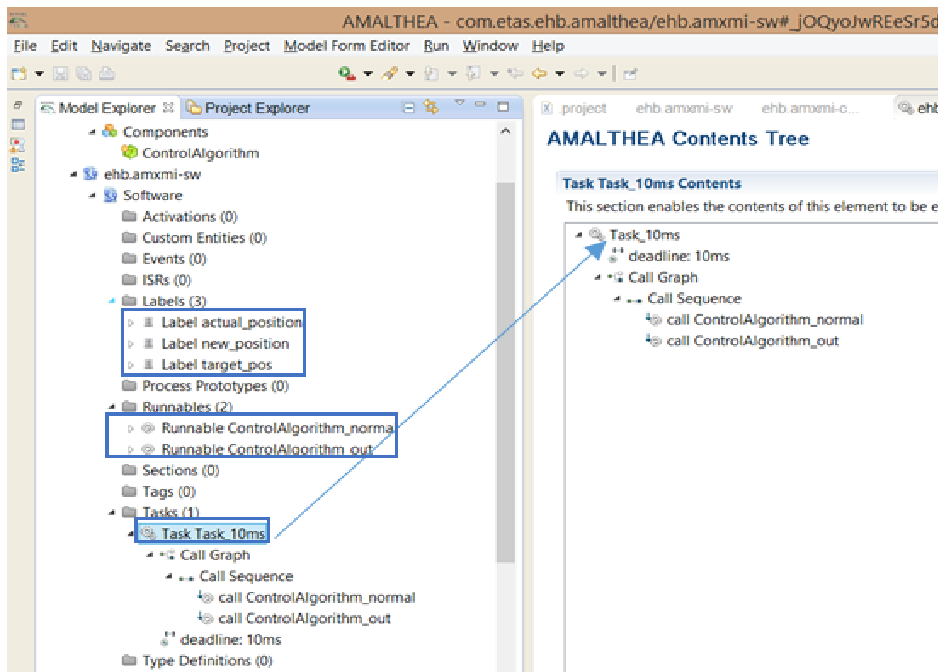

Figure 3.5: The software architectures of the case study (Tasks, Runnables and labels) that are defined using AMALTHEA environment

In the mapping step, the defined software (tasks and runnables) is allocated to the described multicore-system and the related data (like labels) is mapped to the memory (look at figure 3.6). This distribution depends also on the selected variants.

The operating system model defines the schedule of the tasks on the cores. Moreover it assign the components, which describe the behavior of software components and are modeled in ASCET environment, to the related software components in AMALTHEA.

The last step is generating the glue C code in addition to the needed configurations (like operating system configuration, Real-Time Environment (RTE) configuration etc..)from the AMALTHEA models (components model, software model, hardware model, operating system model etc..). This code extends the ECU software to support the multi-core platform.

This glue code will be integrated with the component implementation, which is performed parallel using ASCET.

Finally, timing analysis will be performed and the results will be integrated in the repository of AMALTHEA. Consequently, another iteration with other requirements and variants can begin.

3.3.2 ASCET modeling process

Bosch uses ASCET to model the executable specification, which may depend on various requirements, like the limitations of the microcontroller in terms of space or real-time capability. These executable specification will be modeled using components that are independent of the target system, which ensures the re-usability of these components.

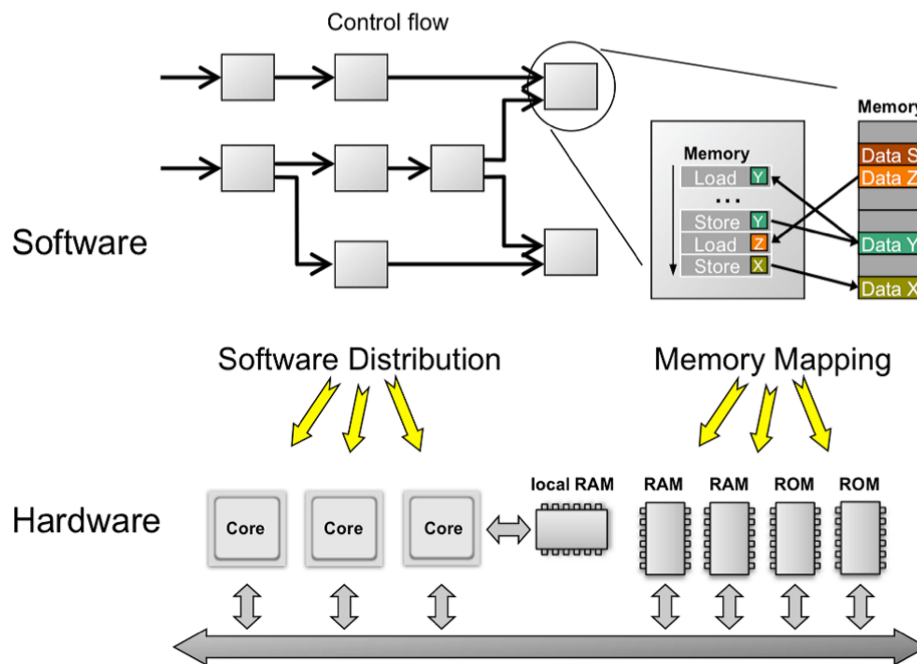

Figure 3.6: The software distribution and memory mapping in AMALTHEA from [33]

The modeling process will be achieved using the viewpoints and editors of ASCET-MD (the block diagram editor, the finite state machine editor, ESDL editor and C code editor).

According to the Bosch case study scenario the block diagram will be used for modeling the control functions. This diagram uses multiple blocks and represents both data flow and control flow between the different blocks. Each block models the behavior of the software component and can import/ export data, which is represented as messages, from/ to other blocks.

For example figure 3.7 shows the implementation of the ControlAlgorithm software component, which is defined in AMALTHEA as explained in section 3.3.1, using the type AscetModule. The purpose of the algorithm is to calculate the new position of the vehicle according to its actual position and the desired position.

Therefore, it defines both target position and the actual position as input whereas the new position as output. The values of the input will be imported from other blocks in the term of input messages (target_pos and actual_position). Similarly, the output (new_position) will be exported to other components as output message. During modeling the behavior of this algorithm various pre-defined components can be used, for example Limiter component provided from ASCET database and PIDT1 component whose functionality is described in other block diagram. Moreover, the different types of messages, parameters and variables can be followed between the components and also manipulated using different available arithmetical and logical calculations.

After modeling of the different components, ASCET will generate optimal C code, which implements the functionality of the component. As a result, this component can be used by integrating this code on the ECU.

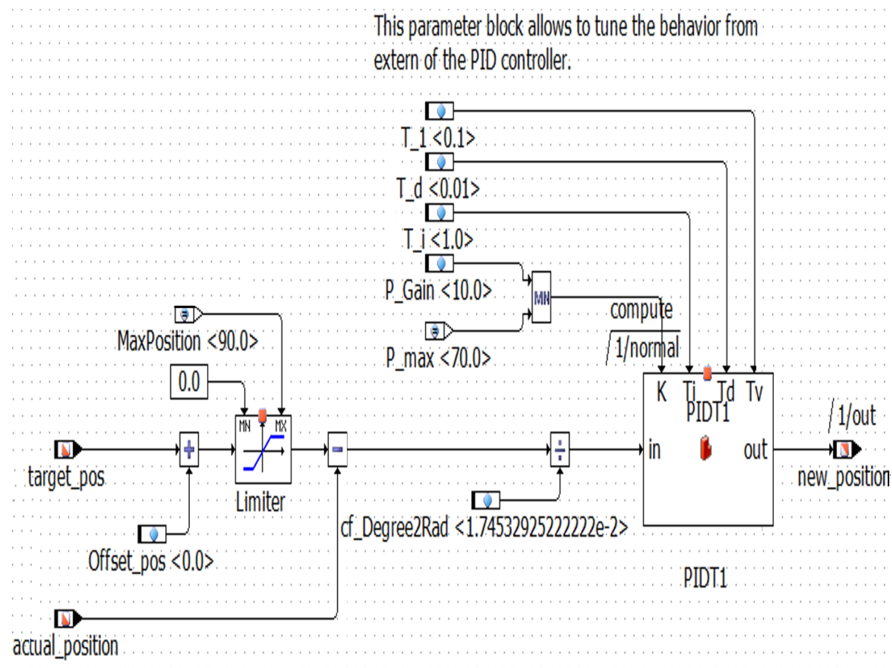

Figure 3.7: Modeling of control algorithm in ASCET using block diagram editor

3.3.3 The integration of the ASCET and AMALTHEA models

The generated C code of AMALTHEA, which defines the software architecture in addition to other information about the hardware, scheduling etc., will be integrated with the generated C code of ASCET, which defines the functionality of the software components, on the circuits. If there are inconsistencies, they will be detected too late, when the errors occurred by compilation and linking stages. Examples of the inconsistencies are the syntactical errors, like using different spelling to type the name of Runnable in AMALTHEA and its Implementation Method in ASCET, and the semantic errors, like the case that there is no defined implementation method for a defined Runnable.

Correcting these errors needs a lot of time to be fixed because the translation process may take several hours and must be restarted after the correction of each error. Moreover, some of the inconsistencies could not be detected in this stage, which means that more time and cost will be needed to resolve them.

3.4 The consistency problem in Bosch case study

According to the case study of Bosch we can see that Bosch uses different models to develop the automotive system. These models are separately constructed and maintained and also share the same semantics. For example the component is described using AMALTHEA standard and its behavior is described using ASCET. This correspondence has to be insured during the development in order to avoid inconsistency after integrating the different models.

Bosch developers depend on information exchange along the process development in order to develop consistent models. Therefore they exchange Extensible Markup Language (XML) files to share the information.

For example the information about each software component developed in AMALTHEA (like name of component, its internal architecture and imported/ exported interfaces) has to be written by hand and shared with the developers of ASCET, who describe the functionality of these software component.

The related information (like the name of the software component in AMALTHEA and the name of ASCET module, which describes the functionality of the software component) can be distributed over multiple documents and reused through copy and paste techniques. The changes in individual artifacts can lead to multiple updates in different documents whereby it is difficult to determine which documents have been affected.

Each modification of the specification documents as well as models have to be synchronized to keep the consistency. The synchronization process is performed also manually and costs time, because it has to be done between each pair of models.

The consistency of these models cannot be currently checked during the modeling process. That means the conflicts, that may be caused by development, synchronization or because of unknown correspondences hidden in tools, will be discovered only after integrating the C code generated from AMALTHEA and ASCET. As mentioned in the last section 3.3.3 the inconsistency between the models will cause compilation and linking errors, which take a long time to be fixed, because each translation may take several hours. What exacerbates the problem, is that not all conflicts can be detected using the compiler /linker. The later the conflicts are detected, the more effort and cost are needed. Additionally, the conflicts resolution after generating the code will cause drift and erosion between the models and the implementation.

Hence, we can see that the consistency preservation between Bosch's models is an expensive process. This is mainly because the time needed for synchronization and resolving the conflicts in the implementation stage.

Thereby, there is a need to reduce this effort and cost through applying more efficient mechanism to detect the inconsistent cases automatically, synchronize the models and fix the conflicts in earlier stages.

3.5 Research question

This study aims to improve the automotive system development process to ensure the consistency of the heterogeneous used models with lowest possible cost. The current automotive software development process suffers that the automotive models that share semantic overlaps are developed independently of each other using special tools. The fragmentation and redundancy of the information between these models can lead to inconsistency. The correspondences between the automotive metamodels are nevertheless neither explicitly defined nor suitable for all projects. Therefore, there is need to express these correspondences declaratively in high level of abstraction, like the declarative expression using MIR language. Defining these correspondences will enable the automatic detection of the inconsistent cases in the modeling stage and will allow applying automatic

synchronization mechanism based on these correspondences. To our knowledge, there is no efficient mechanism to detect the potential inconsistency and resolve it fully automatically. Vitruvius view-based development concept offers a gut mechanism for the automatic synchronization, from which this work aims to benefit. However, the automotive software development is performed using different domain-specific modeling tools, which are not supported by Vitruvius. Besides that, the current Vitruvius prototype does not support the integration of two or more legacy models that are developed using the existing external tools. This raises the main research question: *How can Vitruvius concept be extended in order to be used for improving the automotive software development?* This main question can be divided into the following research questions:

- ***RQ1:*** Can MIR language express the correspondences rules between the automotive metamodels in abstract form? If not, what is missed and has to be developed?
- ***RQ2:*** How should the development process of Vitruvius be evolved in order to support the automotive systems development? What are the most prominent activities that should be included?
- ***RQ3:*** How can the scenario of "legacy" models be supported by Vitruvius? I.e. how can two or more related legacy models be integrated in change-based development?
- ***RQ4:*** How can Vitruvius approach be applied when the used models are developed and updated using other tools, especially when these tools are not open source tools and cannot be integrated with Vitruvius prototype?
- ***RQ5:*** Can the effort needed to detect and resolve potential inconsistencies be reduced by applying Vitruvius approach?
- ***RQ6:*** How far can the automatic consistency preservation between automotive systems be achieved?

4 Applying Vitruvius approach in automotive systems development

This chapter describes the main research objectives in section 4.1. Section 4.2 discusses using MIR for expressing the correspondences between the automotive metamodels. The main new activities that have to be included in Vitruvius development process are described in section 4.3. Section 4.4 illustrates mechanism for integrating two and more related legacy models in Vitruvius. Section 4.5 defines two scenarios to apply Vitruvius concept when the legacy models are developed using external modeling tools.

4.1 Research objective

To answer and evaluate the research questions I have divided my work into the following objectives:

- ***RO1:*** Evaluate MIR language through detecting the consistency rules between the standards used by Bosch (AMALTHEA, ASCET and SysML) and express them by MIR in abstract level. Then validate and test MIR's features and abilities and suggest solution, new features or updates if needed.

- ***RO2:*** Investigate whether the default steps of Vitruvius development process can be followed to apply this approach in practice (like the case by automotive system development). If not, describe the additional steps.

- ***RO3:*** Define and evaluate a specific algorithm to support the scenario of integrating several related legacy models in Vitruvius development process. This requires to detect and link the correlated artifacts of the models, check the consistency and resolve the conflicts between them. Then evaluate the defined algorithm by the case-study of Bosch.

- ***RO4:*** Define the required conditions of using external modeling tools with change-based development approach and describe how we can achieve them to benefit from Vitruvius' features.

  Present strategy that can facilitate the using of Vitruvius platform to preserve the consistency between the models, which are built and developed outside Vitruvius environment using external tools, like closed-source modeling tools.

- ***RO5:*** Assess the extended Vitruvius prototype in practice via a case study of automotive system development through evaluating two features: the consistency check

and conflicts resolution. Compare the result with traditional method (exchange XML files) to determine the efficiency of the proposed solution.

- ***RO6:*** Determine whether Vitruvius can achieve the automatic consistency preservation. If it cannot, list the cases which need a review and feedback from developer

4.2 Using MIR language to declare the consistency rules

The first research objective RO1 aims to evaluate the expressing of the correspondence rules by MIR language. The main purpose is to evaluate the capabilities and features of MIR by the case study and conclude whether it can be used in Vitruvius development process. The following section 4.2.1 explains the benefits of using this language during applying Vitruvius approach in automotive systems development. Moreover, in section 4.2.2 I will illustrate which parts of MIR are evaluated and which points are considered by the evaluation. The problem found by evaluation, the suggested solution and the ideas of developing MIR will be discussed in the section 4.2.3

4.2.1 The reason of using MIR language

The consistency preservation of the automotive system (as large model-based system) lacks to the formalism that declares the correspondence rules in an abstract, easily readable and more comprehensible form. The declarative defining these rules will help to avoid the potential inconsistency between the models. Additionally, it will be used as a basis for adopting automatic consistency preservation approach instead of the manual used one.

The MIR language (described in section 2.4) is compatible with Vitruvius framework and meets the abovementioned requirements. This language defines the mapping between the artifacts, the consistency invariants and the appropriate modification response action, which can be executed when the invariants are violated.

The main feature of MIR is that it is designed to be able to generate automatically the bidirectional transformations between the metamodels from the declarative expression of the correspondence rules between them. These transformations can be used by the synchronisation or by resolving the conflicts between the models.

As a result, the developers will not need to deal with the technical details of these transformations because they are separated from abstract synchronization logic. Moreover, the declarative expression of the individual rules and actions will ease the reuse within projects with identical or similar metamodels.

For the abovementioned reasons, I choose MIR to declare the correspondence between the automotive system as a first step of applying Vitruvius in automotive system development on one hand and to evaluate its potential by a practical case study for the purpose of improving it in the other hand.

4.2.2 The evaluation mechanisms

MIR is still under development. Actually the editor of MIR supports only the declaration of mapping. Therefore both of the invariants and responses are not evaluated. Declaring the

mapping between elements can be done using the mapping blocks (which are described in section 2.4.1 and shown in listing 1). These blocks are *when-where* block, which determines the pre- and post- conditions of this mapping, *with-block* block, which defines computed values for the mapped attributes, and *with map* which declares the sub-mapping.

The example illustrated in listing 3 describes the mapping between the runnables in AMALTHEA (type of Runnable) and the processes in ASCET (type of Method). In this example the *when-where* block restricts the mapping with the condition that the method in ASCET must have neither arguments nor return values.

This restriction can be expressed as two *when-where* conditions. The first one is *equals(pr.ret, null)*, which is checked as the pre-condition*(pr.net == null)* by the mapping from ASCET to AMALTHEA (from left to right) and enforced as the post-condition (pr.net = null) by the mapping from AMALTHEA to ASCET (from right to left). The second condition is*empty(pr.arguments)*, which will be translated to a *pr.arguments. isEmpty()* check by the mapping from ASCET to AMALTHEA (from left-to-right) and *pr.arguments.clear()* enforcement by the mapping from AMALTHEA to ASCET (from right to left). The *with map* defines the sub-mapping between the attributes.

```
map adom.Method as pr and sw.Runnable as runn
 {
           when-where {empty(pr.arguments)
                          equals(pr.ret,null)}
           with map pr.(name) and runn.(name)
 }
```

Listing 3: The MIR definition of the correspondence between runnable and process(method)

The evaluation of MIR language ability of express the correspondences considers the following points: the ability of defining the mapping between the artifacts of the automotive metamodels, the level of the abstraction, readable and clarity of the defined declarative mapping, and the validity of the generated bidirectional transformations.

To achieve this evaluation the correspondence rules between the metamodels of the case study are defined and written using the current version of MIR. The found problems by defining the mapping or by the readability of the rules in addition to some suggested solutions to solve them will be discussed and illustrated by examples in section 4.2.3.

4.2.3 The problems by MIR language and the suggested solutions

The current version of MIR have some obstacles that hinder the use of MIR in automotive system development using Vitruvius approach. In this section I will illustrate the problems of MIR with examples and I will suggest solutions and improvement ideas.

4.2.3.1 Referring to any order of objects

The correspondence rules can be nested to declare the sub-correspondences between the attributes or the references. That requires the ability to refer to these references in order

to map them. In some cases the access of these references is not obvious and depends on calling different order of instances.

This problem can appear in some structures like composite pattern design. This design defines simple objects and a complex composite objects and treats them similar (as it shown in figure 4.2.3.1). If the mapping between the *Composite1* class for instance and another class is declared using MIR, then the description of the sub-mapping or constraints related related to Class B cannot be defined. That is because the referring to an object of Class B can be done only through calling unknown arrangements of instances of *Composite1* class and *Composite2* class.

The following examples show some of the multiple ways to refer to an object of class B that can be found:

$$comp1.comp2.leaf.b,$$
$$comp1.comp1'.leaf.b,$$
$$comp1.comp2.comp1'.comp2'.leaf.b,$$
$$comp1.comp1'.comp2.comp2'.leaf.b, \ldots$$

where $(comp1, comp1')$ instances of Composite1, $(comp2, comp2')$ instances of Composite2 and b is instance of B.

The expression of any order of objects is not supported by MIR. To make this problem much clearer I will illustrate it by the following concrete example of the case study.

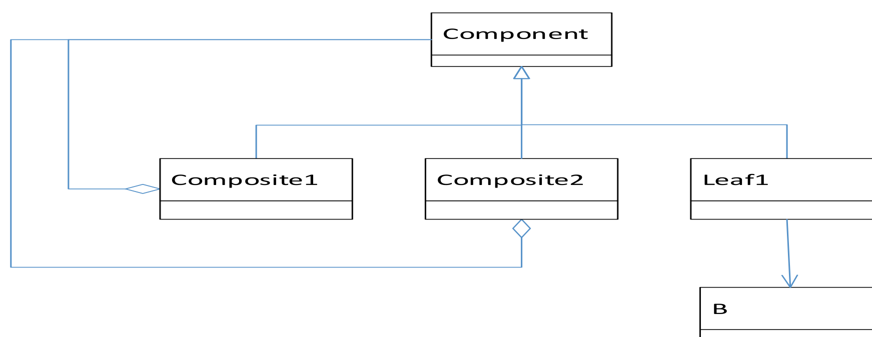

Figure 4.1: Composite pattern structure illustrates the multiple ways to refer to on object of class B.

Example: Both the task type in ASCET and the task type in AMALTHEA are correspondent with each other. This correspondence is nested to include the runnables of the Task in AMALTHEA and their implementation processes defined in the ASCET task. To illustrate this sub-correspondence I will explain the structure of the Task type in both metaclasses.

The Task in ASCET has multiple processes. The processes are type of Method and have neither arguments nor return type (see figure 4.4). These processes describe the functionality of the runnables, which are described in AMALTHEA. Therefore the correspondences between the processes (type of Method) and runnables in AMALTHEA have

to be defined during the definition of the correspondence between the Task metaclasses. It is easy to refer to the methods in ASCET (as shown in 4.4), however it is not the case in AMALTHEA. Each Task in AMALTHEA, which is a subclass of Process class, defines its behavior during the execution, which includes the calling of various items. The runnables are part of these called items. The calling of items differs according to the task and can be one of the following types or mixing of them see figure 4.2.

- Call a sequence of elements (using **CallSequence** class). This type determines a list of elements that are called to be executed by the Process.
- Call one of different defined execution paths according to either value of a label (using **LabelSwitch** class) or a probability-value (using **ProbabilitySwitch** class). This kind is like if-else or switch-case statements in a programming language and requires defining the values related to each path (when **LabelSwitch** class is used) or the probability of executing each path (when **ProbabilitySwitch** class is used).

So, to know the runnables which belong to a task all the elements called from all the possible execution paths have to be selected and then filtered according to the type call-Runnable (figure 8.4 shows example of the different types of execution paths). As we have seen in this example, the referring of all the runnables, which are belong to a task depends on different unknown order of instances (**CallSequence**,**LabelSwitch** and **ProbabilitySwitch**), which is impossible to be defined as a declarative general correspondence rule using the current version of MIR.

Suggested solution: The suggested solution depends on referring to any order of objects with new defined keyword like three successive points "**...**", which can be defined in the Xtext grammar with the help of Xbase expressions. For this purpose the Xtext code defines and calls a recursive Xbase method, which checks all the possible nested paths that can lead to the target objects through calling different orders of objects and collects these objects using collection, filter and aggregation lambda operations. As a result, the correspondence rules will be described in abstract level and the different orders of object will be handled internally during the translation of the rule.

Listing 4 shows the applying of this solution by the mapping between Task metaclass of AMALTHEA (shown in figure 4.2) and Task metaclass of ASCET (shown in figure 4.4).

```
map adom.Task as at and sw.Task as st
 { with map at.(name) and st.(name)
      with map at.(priority) and st.(priority)
      with map at.(processes as pr ) and  st.(callGraph)... (graphEntries[sw.
          CallSequence]).(calls[sw.TaskRunnableCall]).(runnable as runn)
      {     when-where {empty(pr.arguments)
                         equals(pr.ret,null)}
            with map pr.(name as n1) and runn.(name as n2)
                   } }
```

Listing 4: Declare the correspondence rule between Task in AMALTHEA and Task in ASCET Using "..." to refer to any order of objects

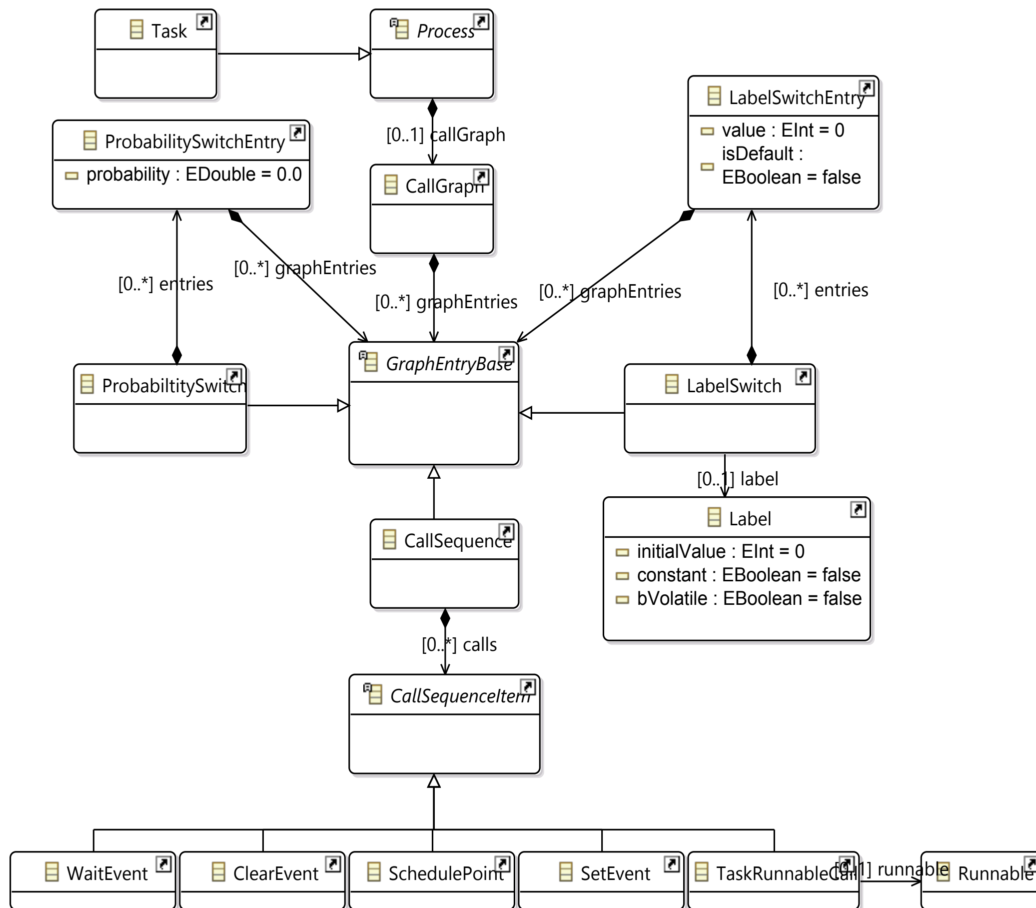

Figure 4.2: Example of referring to object (**TaskRunnableCall**) using undefined order of objects (**LabelSwitch**, **ProbabilitySwitch** and **CallSequence**)

4.2.3.2 One-to-many mapping

In some cases one element is corresponding with multiple elements in the contrast metamodel. For example, by the inheritance the subclasses may have the same correspondence of their superclass. This means, they will be correspondent with the same element in the contrast metamodel (see figure 4.3). Hence, it is necessary to determine with which one of these classes the related element in the corresponding metamodel has to be mapped. The description of this corresponding rule using MIR language requires repeating similar rules for each subclass to illustrate that all subclasses have to be mapped to the corresponding element. Nevertheless, the mapping is unclear, because MIR language does not offer a method to determine the target element, which has to be chosen as a default choice for the inverse mapping.

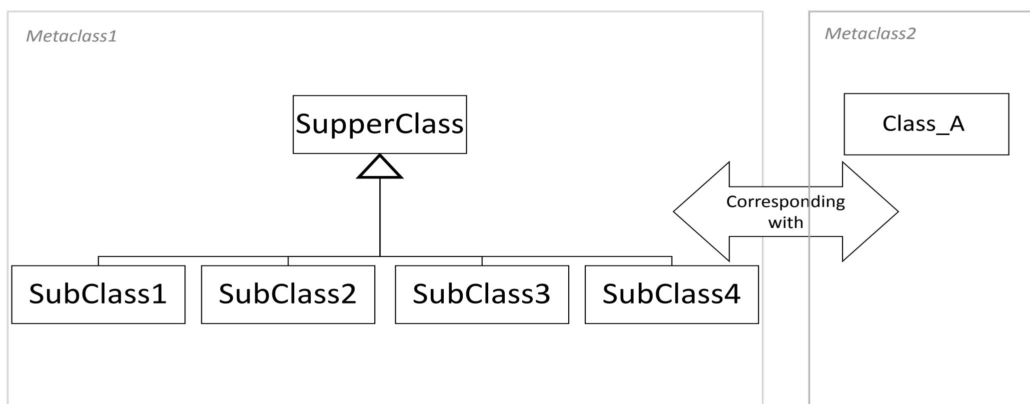

Figure 4.3: Example of mapping one-to-many relation

Examples: In this example I will explain how the type Task in ASCET and some of its subtypes are corresponding with the type Task in AMALTHEA. The type Task in ASCET metamodel has five subtypes (InitTask, SoftwareTask, PeriodicTask, TimeTableTask and InterruptTask) see figure 4.4. Wherase there is only one Task type in AMALTHEA metamodel shown in figure 4.5.

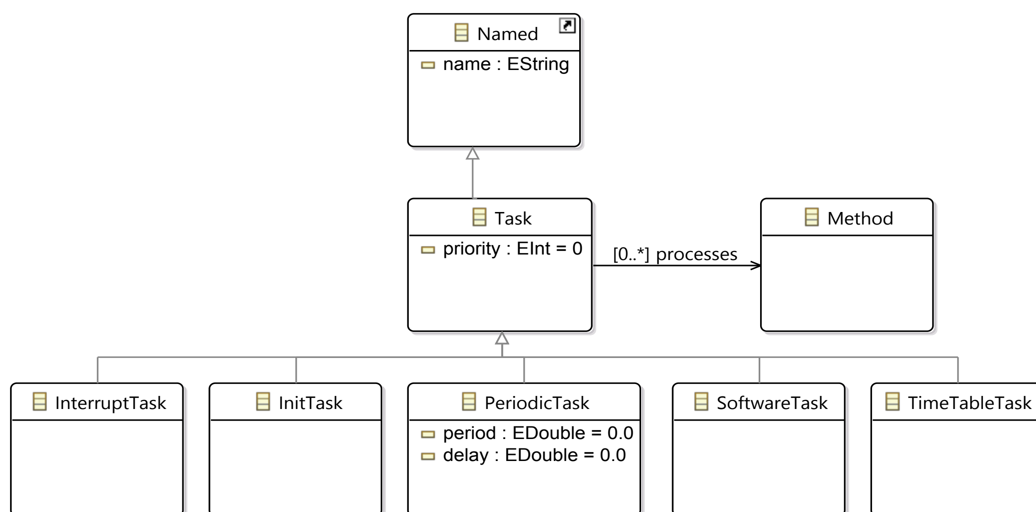

Figure 4.4: The task type in ASCET

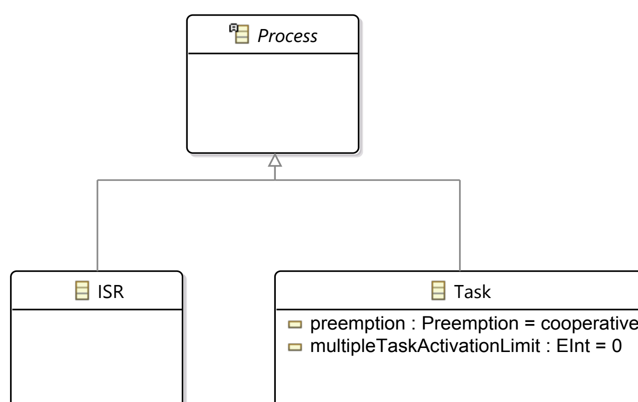

Figure 4.5: The Task and ISR types in AMALTHEA

The Task type in ASCET is corresponding with Task type in AMALTHEA. This correspondence is inherited partly. In other words, all the subtypes of ASCET Task (except InterruptTask) are corresponding with Task type in AMALTHEA metamodel. Declaring these correspondences with MIR requires writing five similar correspondences (one for the supertype and four for the subTypes). Moreover, it is ambiguous to which one of these five types the Task type in AMALTHEA has to be mapped.

The listing 5 shows the similar MIR mapping blocks between Task type in AMALTHEA on one hand and the Task type, InitTask type, SoftwareTask type and TimeTableTask type from ASCET on the other hand.

The last mapping block describes the additional needed mapping between the subtype InterruptTask in ASCET and ISR type (not Task type) in AMALTHEA.

```
generates package mir.ascet2amalthea
generates type ASCET2AMALTHEA

import package "http://www.amalthea.itea2.org/model/1.1.0/sw" as sw
import package "http://amalthea.itea2.org/model/1.1.0/components" as components
import package "http://com.bosch.swan.ascet.adom/1.0" as adom

map adom.Task as at and sw.Task as st
 {
      with map at.(name) and st.(name)
      with map at.(priority) and st.(priority)
      with map at.(processes as pr ) and
      st.(callGraph)...(graphEntries[sw.CallSequence]).(calls[sw.TaskRunnableCall])
          .(runnable as runn)
      {
            when-where {empty(pr.arguments)
                          equals(pr.ret,null)}
            with map pr.(name as n1) and runn.(name as n2)
      }
 } //==========================================
map adom.InitTask as ait and sw.Task as st
 {
      with map ait.(name) and st.(name)
      with map ait.(priority) and st.(priority)
      with map ait.(processes as pr) and
            st.(callGraph)...(graphEntries[sw.CallSequence]).(calls[sw.
                TaskRunnableCall]).(runnable as runn)
      {
            when-where {empty(pr.arguments)
                          equals(pr.ret,null)}
            with map pr.(name) and runn.(name)
      }
 } //==========================================
 map adom.SoftwareTask as ast and sw.Task as st
 {
      with map ast.(name) and st.(name)
      with map ast.(priority) and st.(priority)
      with map ast.(processes as pr) and
      st.(callGraph)...(graphEntries[sw.CallSequence]).(calls[sw.TaskRunnableCall])
          .(runnable as runn)
      {
            when-where {empty(pr.arguments)
                          equals(pr.ret,null)}
            with map pr.(name) and runn.(name)
      }
 } //==========================================
 map adom.TimeTableTask as attt and sw.Task as st
```

```
 {
      with map attt.(name) and st.(name)
      with map attt.(priority) and st.(priority)
      with map attt.(processes as pr) and
      st.(callGraph)...(graphEntries[sw.CallSequence]).(calls[sw.TaskRunnableCall])
          .(runnable as runn)
      {
            when-where {empty(pr.arguments)
                          equals(pr.ret,null)}
            with map pr.(name) and runn.(name)
      }
} //============================================
  map adom.PeriodicTask as apt and sw.Task as sisr
 {
      with map apt.(name) and sisr.(name)
      with map apt.(priority)and sisr.(priority)
      with map apt.(processes as pr )and sisr.(callGraph)...(graphEntries[sw.
          CallSequence]).(calls[sw.TaskRunnableCall]).(runnable as runn)
      {
            when-where {empty(pr.arguments)
                          equals(pr.ret,null)}
            with map pr.(name) and runn.(name)
      }
 with-block
{
switch sisr. stimuli[sw.Periodic].offset.unit
{
case TimeUnit.ps : apt.delay=sisr.(stimuli[sw.Periodic]).offset.value / 10^12
case TimeUnit.ns : apt.delay=sisr.(stimuli[sw.Periodic]).offset.value / 10^9
case TimeUnit.us : apt.delay=sisr.(stimuli[sw.Periodic]).offset.value / 10^6
case TimeUnit.ms : apt.delay=sisr.(stimuli[sw.Periodic]).offset.value / 10^3
default : apt.delay=sisr.(stimuli[sw.Periodic]).offset.value
}
}
switch sisr. stimuli[sw.Periodic].Recurrence.unit
{
case TimeUnit.ps : apt.period=sisr.(stimuli[sw.Periodic]).Recurrence.value / 10^12
case TimeUnit.ns : apt.period=sisr.(stimuli[sw.Periodic]).Recurrence.value / 10^9
case TimeUnit.us : apt.period=sisr.(stimuli[sw.Periodic]).Recurrence.value / 10^6
case TimeUnit.ms : apt.period=sisr.(stimuli[sw.Periodic]).Recurrence.value / 10^3
default : apt.period=sisr.(stimuli[sw.Periodic]).Recurrence.value
}
} } //============================================
map adom.InterruptTask as ait and sw.ISR as sisr
 {
      with map ait.(name) and sisr.(name)
      with map ait.(priority) and sisr.(priority)
```

```
    with map ait.(processes as pr) and
    sisr.(callGraph)...(graphEntries[sw.CallSequence]).(calls[sw.TaskRunnableCall
        ]).(runnable as runn)
    {
          when-where {empty(pr.arguments)
                        equals(pr.ret,null)}
          with map pr.(name) and runn.(name)
    }
}
```

Listing 5: The similar correspondence rules between Task and its subclasses in ASCET and Task in AMALTHEA

The suggested solution:

- The first solution for one-many mapping problem, is to determine one default corresponding rule, which has to be executed for the inverse mapping. To achieve this goal the keyword *"default"* can be added to the MIR language, otherwise the developer has to be asked every time to choose the appropriate element for the mapping. The mechanism of asking the developer has to be supported from MIR too.
- A more efficient solution for the inherited corresponding rule is to define only one correspondence rule (between the superclass and the corresponding class) using the following keywords:
  - the keyword *"inherited"* to indicate to the inherited correspondences.
  - the keyword *"default"* with the name of the default class for the inverse mapping.

  The following listing shows the applying of the suggested solution to the abovementioned example 4.2.3.2.

```
inherited map adom.Task default adom.SoftwareTask as at and sw.Task as st
 {
      with map at.(name) and st.(name)
      with map at.(priority) and st.(priority)
      with map at.(processes as pr ) and  st.(callGraph)...(graphEntries[sw.
          CallSequence]).(calls[sw.TaskRunnableCall]).(runnable as runn)
      {
            when-where {empty(pr.arguments)
                          equals(pr.ret,null)}
            with map pr.(name as n1) and runn.(name as n2)
      }
 }
```

Listing 6: Applying of the suggested solution to the abovementioned example, using *inherited* and *default* keywords 4.2.3.2

When the *default* keyword is not used, the developer has to be asked every time to identify whether the superclass or one of the subclasses will be used for the one-many mapping. Moreover this solution suggests declaring the rules of the subclasses, which do not inherit the correspondence rule of the superclass, separately. For example in the abovementioned example the correspondence between the type InterruptTask in ASCET (subtype of Task) and its corresponding type in AMALTHEA (ISR instead of Task) has to be declared once again separately as it has been shown in the listing 7.

```
map adom.InterruptTask as ait and sw.ISR as sisr
 {
      with map ait.(name) and sisr.(name)
      with map ait.(priority) and sisr.(priority)
      with map ait.(processes as pr) and
      sisr.(callGraph)...(graphEntries[sw.CallSequence]).(calls[sw.
          TaskRunnableCall]).(runnable as runn)
      {
            when-where {empty(pr.arguments)
                          equals(pr.ret,null)}
            with map pr.(name) and runn.(name)
      }
 }
```

Listing 7: Applying of the suggested solution to the abovementioned example 4.2.3.2 will define additional rules for the non-inherited rules

In this example the mapping between PeriodicTask and Task is nested to define the relation of the time constraints (recurrence and offset), therefore it will be also separately defined in order to describe the additional sub-mapping. This rule will be discussed in details in the section 4.2.3.3.

4.2.3.3 Duplicate writing similar rules

The main objective of using MIR language is to describe the correspondence rules in high level of abstraction and readable way. Therefore MIR has to avoid repeating similar rules, which can not be well read or distinguished from each other.

In some cases, several similar rules can be repeated in order to reflect small changes (like values of the attributes) occurred because of different possible changes in the pre-conditions defined in *when-where* block.

For example, by the mapping between attributes that have different measurement units it is necessary to convert the values of these attribute according to certain unit defined as pre-condition in *when-where* block and assign them to the related attributes using *with-block* (mentioned in section 2.4.1). The changes of the measurement units lead to changes in the attributes' values. As a result, similar rules will be repeated to reflect the change of the attribute's value according to the selected measurement unit.

Example: For example, the PeriodicTask in ASCET and Task in AMALTHEA are correspondent with each other. Defining the mapping between them requires defining the

sub-mapping between the temporal attributes too. The PeriodicTask has two attributes periodic and delay. The related information of these attributes are stored in *Periodic* class in AMALTHEA, which the Task class refers to (for more information see stimulus model 8.2.1.2 and the software model structure that illustrates the relation between Task and Stimulus in the appendix 8.1).

The attribute **recurrence** in AMALTHEA is correspondent with the attribute **periodic** in ASCET and the attribute **offset** in AMALTHEA is also correspondent with the attribute **delay** in ASCET.

Depending on the attribute **TimeUnit** the value of both the attributes **recurrence** and **offset** in AMALTHEA can be in seconds, milliseconds, microseconds, nanoseconds or picoseconds, whereas the values of the attributes **period** and **delay** are always saved in seconds.

Defining the sub-mapping have to convert the attributes' values according to their units.

As aforementioned MIR allows to determine the time unit of each of these attributes as a pre-conditions using *when-where* block and assign the converted value to the related attributes using the *with-block*.

In each mapping block we can determine only one time unit for each of the **recurrence** and **offset** attributes. This means, we need to declare **25** rules in order to cover the all possible cases needed to describe the correspondence between PeriodicTask and Task.

That is because these attributes are independent of each other and each one may have one of five measurement unit. Therefore, there are (5 multiple by 5) different possible pre-conditions as well as required mapping blocks. In the following listing 8 I will show an example of the first four rules. The rest 21 rules will be implemented similar with small changes in both *when-where* block and *with-block*. In other words, each one of these similar rules assign the converted values of the attributes *Recurrence* and *offset* using **with-block** according to the measurement units determined in *when-where* block.

```
//The first rule when offset is in second and recurrence in second
map adom.PeriodicTask as apt and sw.Task as sisr
 {
      with map apt.(name) and sisr.(name)
      with map apt.(priority)and sisr.(priority)
      with map sisr.(stimuli[sw.Periodic]).(recurrence).(value) and apt.(period)
      {
            when-where { equals(sisr. stimuli[sw.Periodic].Recurrence.unit, TimeUnit
                .s)}
      }
      with map sisr.(stimuli[sw.Periodic]).(offset).(value) and apt.(delay)
      {
            when-where {equals(sisr.stimuli[sw.Periodic].(recurrence).(unit),
                TimeUnit.s)}
      }
 with map apt.(processes as pr )and sisr.(callGraph)...(graphEntries[sw.CallSequence
    ]).(calls[sw.TaskRunnableCall]).(runnable as runn)
      {
            when-where {empty(pr.arguments)
```

```
                            equals(pr.ret,null)}
            with map pr.(name) and runn.(name)
      }}
//The second rule when offset is in second and recurrence in millisecond
map adom.PeriodicTask as apt and sw.Task as sisr
 {
with map apt.(name) and sisr.(name)
with map apt.(priority)and sisr.(priority)
 with map sisr.(stimuli[sw.Periodic]).(recurrence).(value) and apt.(period)
 {
 when-where {equals(sisr. stimuli[sw.Periodic].Recurrence.unit, TimeUnit.s)}
 }
with map apt.(processes as pr )and sisr.(callGraph)...(graphEntries[sw.CallSequence
    ]).(calls[sw.TaskRunnableCall]).(runnable as runn)
 {
            when-where {empty(pr.arguments)
                            equals(pr.ret,null)}
            with map pr.(name) and runn.(name)
 }
when-where {equals(sisr.stimuli[sw.Periodic].(Recurrence).(unit),TimeUnit.ms)}
With-block
{
      apt.period=sisr.(stimuli[sw.Periodic]).Recurrence.value / 10^3
}}
//The third rule when offset is in second and recurrence in nanosecond
map adom.PeriodicTask as apt and sw.Task as sisr
 {
with map apt.(name) and sisr.(name)
with map apt.(priority)and sisr.(priority)
 with map sisr.(stimuli[sw.Periodic]).(recurrence).(value) and apt.(period)
 {
 when-where {equals(sisr. stimuli[sw.Periodic].Recurrence.unit, TimeUnit.s)}
 }
with map apt.(processes as pr )and sisr.(callGraph)...(graphEntries[sw.CallSequence
    ]).(calls[sw.TaskRunnableCall]).(runnable as runn)
 {
            when-where {empty(pr.arguments)
                            equals(pr.ret,null)}
            with map pr.(name) and runn.(name)
 }
when-where {equals(sisr.stimuli[sw.Periodic].(Recurrence).(unit),TimeUnit. us)}
With-block
{
      apt.period=sisr.(stimuli[sw.Periodic]).Recurrence.value / 10^6
}}
//The fourth rule when offset is in second and recurrence in nanosecond
map adom.PeriodicTask as apt and sw.Task as sisr
```

```
 {
with map apt.(name) and sisr.(name)
with map apt.(priority)and sisr.(priority)
 with map sisr.(stimuli[sw.Periodic]).(recurrence).(value) and apt.(period)
 {
 when-where {equals(sisr. stimuli[sw.Periodic].Recurrence.unit,TimeUnit.s)}
 }
with map apt.(processes as pr )and sisr.(callGraph)...(graphEntries[sw.CallSequence
    ]).(calls[sw.TaskRunnableCall]).(runnable as runn)
 {
           when-where {empty(pr.arguments)
                         equals(pr.ret,null)}
           with map pr.(name) and runn.(name)
 }
when-where {equals(sisr.stimuli[sw.Periodic].(Recurrence).(unit),TimeUnit.ns)}
With-block
{
      apt.period=sisr.(stimuli[sw.Periodic]).Recurrence.value / 10^9
}}
/* The rest 21 rules will be similar,the unit of recurrence and the unit of are
    independent of each other and can take one of the 5 units, therefore the total
    number of possible rules are 5*5. */
```

Listing 8: Example of repeating similar rules according to defined constraints

Other example that illustrates this problem is the description of the correspondence between AscetModule from ASCET and Block from SysML. This example will explained in details in the next section 4.2.3.4 and the repeated similar rules is found in appendix 11.

Suggested solution: Supporting of *if* or *switch* statements can reduce the replication of similar rules. The differences of constraints can be represented as the conditions of *if* or *switch* statements, which control and declare the reflected changes without the need of repeat the other information. As a result of applying the suggested solution the correspondence of the previous example 4.2.3.3 will be declared in only one correspondence rule using two switch statement instead of 25 rules, as it is shown in listing 9.

```
 map adom.PeriodicTask as apt and sw.Task as sisr
 {
     with map apt.(name) and sisr.(name)
     with map apt.(priority)and sisr.(priority)
     with map apt.(processes as pr )and sisr.(callGraph)...(graphEntries[sw.
         CallSequence]).(calls[sw.TaskRunnableCall]).(runnable as runn)
     {
           when-where {empty(pr.arguments)
                         equals(pr.ret,null)
                             }
           with map pr.(name) and runn.(name)
     }
```

```
 with-block
{
switch sisr. stimuli[sw.Periodic].offset.unit
{
case TimeUnit.ps : apt.delay=sisr.(stimuli[sw.Periodic]).offset.value / 10^12
case TimeUnit.ns : apt.delay=sisr.(stimuli[sw.Periodic]).offset.value / 10^9
case TimeUnit.us : apt.delay=sisr.(stimuli[sw.Periodic]).offset.value / 10^6
case TimeUnit.ms : apt.delay=sisr.(stimuli[sw.Periodic]).offset.value / 10^3
default : apt.delay=sisr.(stimuli[sw.Periodic]).offset.value
}
}
switch sisr. stimuli[sw.Periodic].Recurrence.unit
{
case TimeUnit.ps : apt.period=sisr.(stimuli[sw.Periodic]).Recurrence.value / 10^12
case TimeUnit.ns : apt.period=sisr.(stimuli[sw.Periodic]).Recurrence.value / 10^9
case TimeUnit.us : apt.period=sisr.(stimuli[sw.Periodic]).Recurrence.value / 10^6
case TimeUnit.ms : apt.period=sisr.(stimuli[sw.Periodic]).Recurrence.value / 10^3
default : apt.period=sisr.(stimuli[sw.Periodic]).Recurrence.value
}
} }
```

Listing 9: Avoid the repeating of similar rules in example 4.2.3.3 using the suggested *switch* statement

The MIR code in the next section 4.2.3.4 shows also using the suggested *switch* keyword by describing the correspondence between AscetModule from ASCET and Block from SysML.

4.2.3.4 No direct access on a metaclass

By describing of the mapping block, it may be needed to access other artifacts in the structure for the purpose of defining the sub-mapping or to add constraints, pre-conditions that related to the attributes/ references of these artifacts. Sometimes there is no direct access to the desired artifacts.

For example, the unidirectional association, aggregation or composition relations are strictly one way association. That means, the access to the metaclass in the reverse direction of this association cannot be done directly and needs to explore the metamodel in search of the desired metaclass. The figure 4.6 shows three examples of the strictly one way association. Each one illustrates how it is not impossible to access the class **B** directly from the class **A**.

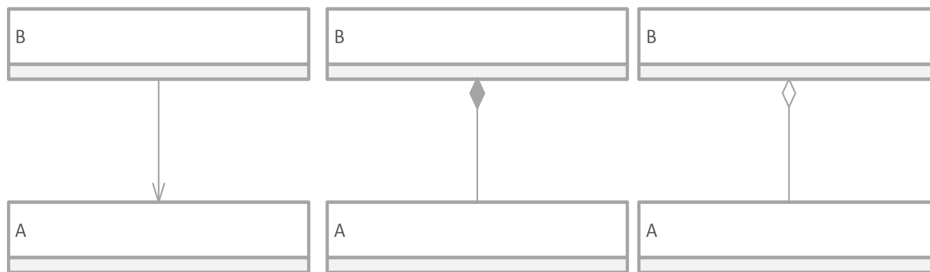

Figure 4.6: Examples of one way association, in each one the metaclass A is not able to access the class B directly

Example: Both SysML and ASCET describe the software components (in term of block in SysML and ASCET module in ASCET) in addition to their interfaces (in term of in/ out flow ports in SysML and read/ write messages in ASCET).

In following I will give an overview about the metaclasses related to the aforementioned semantic correspondence in order to illustrate the problem of declaring this correspondence using MIR.

On one hand each SysML Block metaclass refers to its base-class, which has multiple ports. Each of these ports are encapsulated with the metaclass FlowPort, which has more information about the port like the direction of port (in, out or inout). The figure4.7 shows the relation between Block and FlowPort.

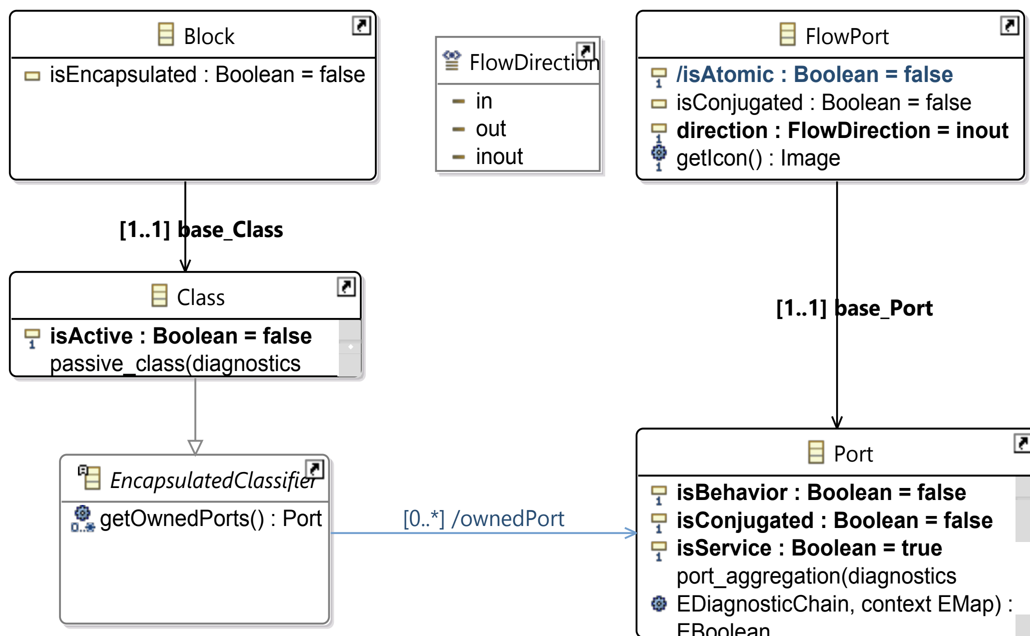

Figure 4.7: Example of no direct access problem

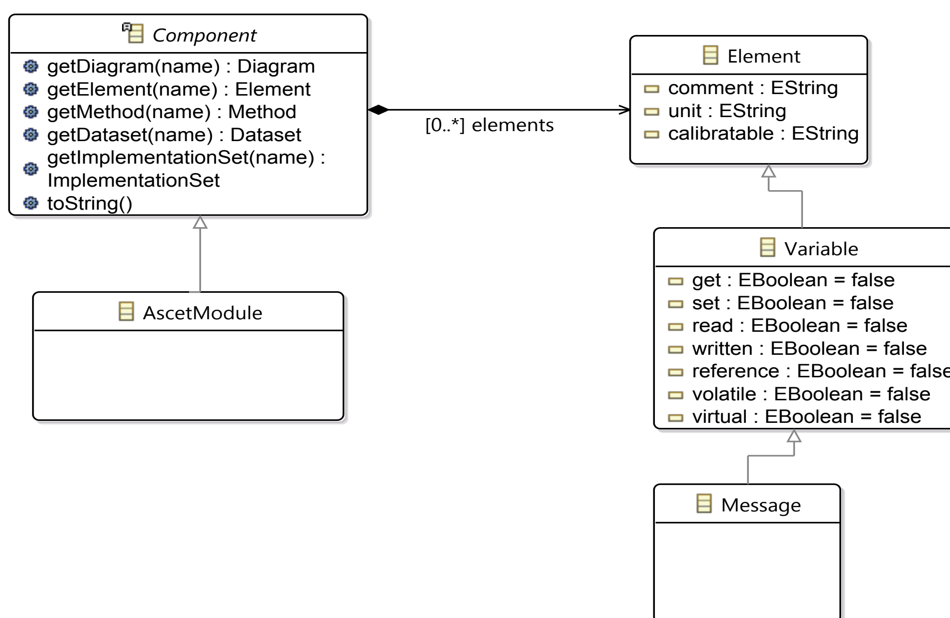

Figure 4.8: The metaclass of AscetModule

On the other hand AscetModule metaclass consists of multiple elements, which can be instance of the type Message. These messages according to their attributes can be classified in one of these types: "message can be only read", "message can be only written" or "message can be read or written" (see figure 4.8).

Defining the mapping between Block and AscetModule requires the ability to access the both of FlowPort and Message metaclasses for the purpose of defining the sub-mapping between them. The Message metaclass can be referred and accessed directly from the AscetModule metaclass whereas the Block metaclass cannot access the FlowPort metaclass directly because of the unidirectional association between them.

Defining this sub-mapping by MIR is not possible because MIR does not support exploring the metamodel in order to inquire about the FlowPort.

Suggested solution: The optimal suggested solution is that MIR Language supports new keywords like **referencedBy**, **composedBy** and **aggregatedBy**. These keywords can ease the declarative expression of the corresponding rules in high level of abstraction. Then these keywords are interpreted internally and generate the mapping depending on Xbase expression.

Defining the appropriate search method for the suggested keywords can be done using lambada expressions, which support the needed aggregation, filter and collection operations.

The other solution is that MIR language enables using either the abovementioned Xbase expressions or OCL constraints, which offers also the required collection and filter operations to explore the metamodel [17].

The following MIR code shows applying the first suggested solution to the aforementioned example 4.2.3.4.

```
generates package mir.sysml2ascet
generates type SYSML2ASCETs

import package "http://com.bosch.swan.ascet.adom/1.0" as adom
import package "http://www.eclipse.org/papyrus/0.7.0/SysML/Blocks" as sysml

map sysml.blocks.Block as bl and adom.AscetModule as ascetModule
{
when-where {equals(bl. isEncapsulated,true)}
with map sysml.PortAndFlow.FlowPort referencedBy bl.(base_Class).(ownedPort) as
    flowport
       and ascetModule.elements(Message) message
{
      with map flowport.(name) and message.(name)
/*The switch block is used as a suggested solution for the problem of repeating
    similar rules. see this example without using switch block in the appendix*/
      with-block {
      switch flowport.(direction)
            case FlowDirection.Out:messag.read= true
                          messag.written=flase
            case FlowDirection.In:messag.read= false
                         messag.written=true
            case FlowDirection.Inout:messag.read= true
                           messag.written=true
     }
}}
```

Listing 10: Using the suggested key **referencedBy** to declare the mapping between SysML Block and AscetModule

4.3 Development process using Vitruvius

This section meets the research objective RO2, which aims to explore the actual development process of Vitruvius and determine whether additional steps may be needed for applying Vitruvius in automotive system development.

Vitruvius Development process according to the methodologist's role can be divided into two stages. The first stage is initialize the environment (create VSUM metamodel, defining view types, defining correspondences between the metamodels, etc.) and it is described in the responsibilities of the methodologist (see section 2.3.1). The tasks of this stage shall be applied in order to use Vitruvius in practice. The first important task is creating VSUM metamodel. VSUM has to be able to represent all the models of the project, for example in the case study of automotive system VSUM metamodel will consist of the legacy SysML, AMALTHEA and ASCET metamodels. The second step is to deduce the semantic correspondences between the metamodels (as it will be described in automotive

system example 5.1.1). These correspondences rules have to be written either with MIR language (see the example in section 5.1) or by using another transformation language like Xtend (as it explained in section 5.3.1). Then these consistency rules are added to the VSUM metamodel. After that the next stage of the development begins. In this stage the developers have to be able to access and modify the view types of VSUM metamodel.

Current version of Vitruvius support only few modeling tools. That means, the developers of the model-based system (like automotive system) have to use existing external modeling tools to build and manipulate their models. As a consequence the development process has to be updated in order to support the following tasks:

- Import a legacy model for one of the following purposes:
  - Generate its related model depending on the integration strategy [29] explained in the section 2.3.3. This step is already performed.
  - Integrate it with the other legacy models. This means finding the correlated elements that belong to different models and linking them together.
  - Update an existing models, which demands deduction of the changes through calculating the differences between the old and new model. Then synchronize the updated models with the other models. This step is colored with green in the use case diagram shown in figure 4.3 to point that this step will be implemented in the future work.
- Check the consistency between the integrated models and resolving the found inconsistent cases, which can be occurred during the modeling using external modeling tools.
- Export the integrated consistent models.

The updated development process of Vitruvius is shown in figure 4.3. In this figure the red use cases indicate to the extension performed by this work, whereas the green use case points to the extension that will be performed by a future work. Moreover shows this figure that the developer uses also an external modeling tool, which allows him to develop the models, export them to Vitruvius environment or import them again for the further development.

The new steps of the development process will be discussed and explained in details in the following sections (4.4,4.5).

4.4 Support the scenario of legacy models

Applying Vitruvius in the automotive system requires the ability of integration the legacy models in Vitruvius development process for the purpose of the reusability and supporting the modeling using external tools during Vitruvius development process. This section propose strategy that meets these requirements and achieves the third research objective RO3 through explaining how the different related legacy models can be integrated in Vitruvius development process.

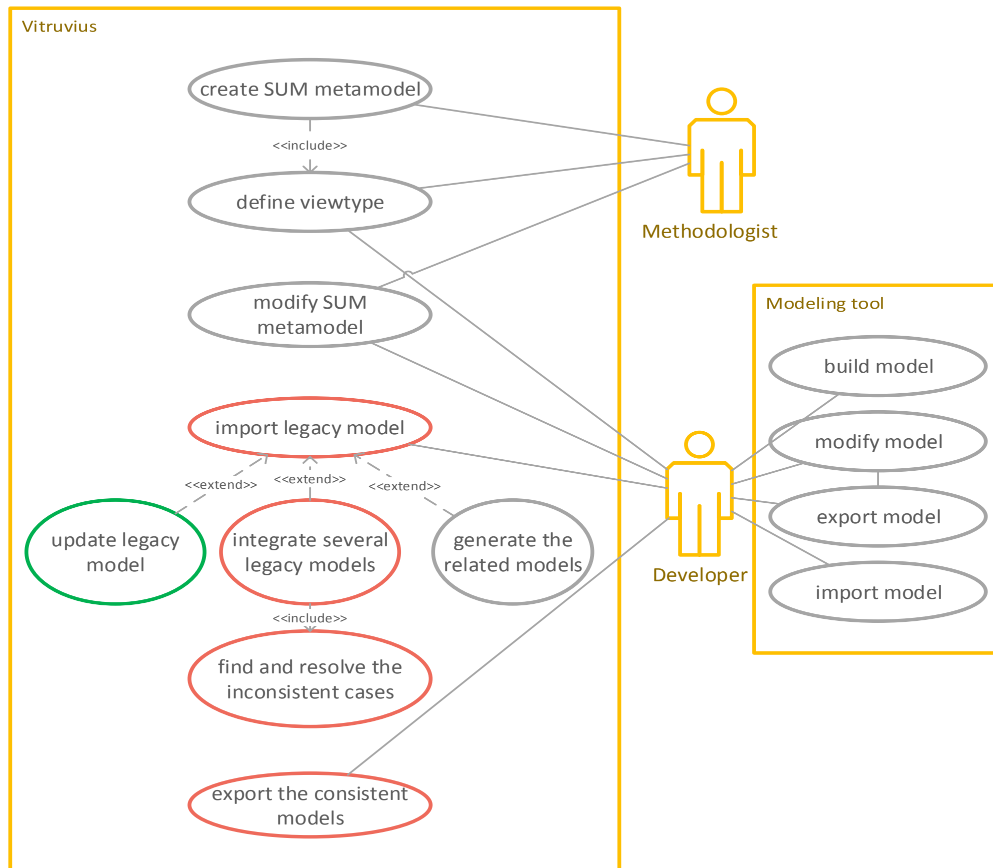

Figure 4.9: The updated use cases for developer roles in Vitruvius. Use cases colored with red are performed in this work. Use case colored with green will be performed in the future work. Old use cases is colored with gray.

As mentioned in section 2.3.3 it is possible to integrate a structural model or object-oriented code into change-based development approach (such as Vitruvius) using RSI and LSI strategies [29]. However, these strategies do not support the integration of two or more models, which share a certain semantics. Therefore I introduce a new strategy to integrate several legacy architectural models in change-based development approach like Vitruvius.

The strategy expects at least two legacy models as input. These models have to be valid and compatible with the defined view types. The pre-conditions of the further delta-based development are that all models under development are consistent with each other and the related elements that belong to these models are linked.

Therefore, the support of legacy models in delta- and view-based development environment like Vitruvius requires linking the related artifacts, the ability of check the consistency and resolving the conflicts between the legacy models.

This work proposes strategy that applies the abovementioned requirements through the following steps (see figure 4.10):

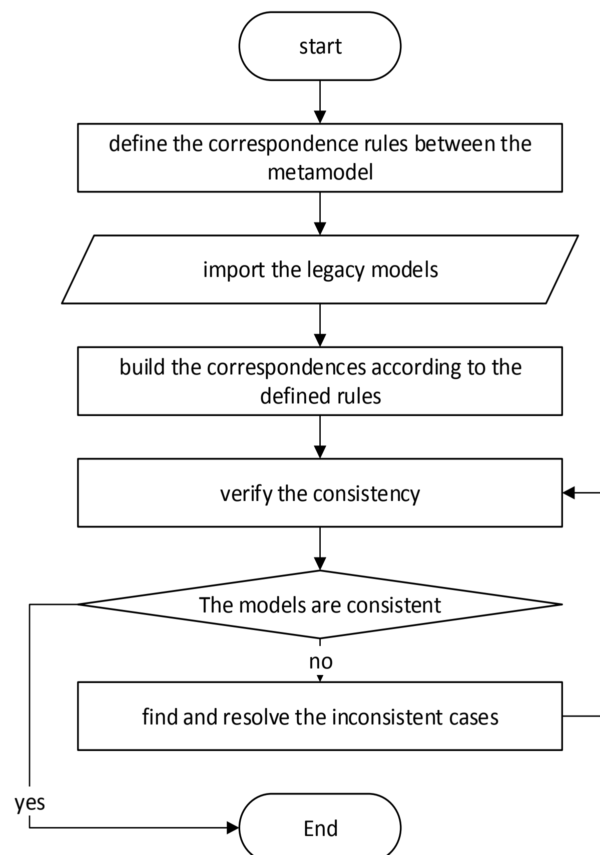

Figure 4.10: Integrate structural models into change-based development approach

- ***S1:*** Describing the correspondences between the metamodels. These correspondences will be used to initialize Vitruvius (creating VSUM metamodel and defining view types that compatible with the legacy models) on one hand and to link the correlated elements on the other hand. The later use of these correspondences rules will be discussed again in the section 4.4.1).

- ***S2*** Import the legacy models and build the correspondences between them according to the rules defined in the first step. These correspondences are needed for further development using change-based development (more details in section 4.4.2).

- ***S3*** Verify the consistency between the artifacts and resolve conflicts and inconsistent cases occurred during the independently development. (read the suggested algorithm in section4.4.3)

The following sub-sections will introduce more details of this strategy.

4.4.1 S1: Describe the correspondences between the metamodels of legacy models

In this step the correspondences between the different elements are defined and the metarepository according to these correspondences are created.

In this section I will suggest two classification of the correspondence rules that ease the implementation of the legacy models integration. Moreover, I will add an additional point that have to be considered by describing the correspondences rules, which is the type of the related attributes and the references of the corresponding elements. I will distinguish and explain two types, which will help to use the correspondence rules in the next step **S2** 4.4.2 to indicate if two artifacts are correspondent or not.

The correspondence rules describe the semantic relation and the consistency between the elements that belong to different metamodels in addition to the semantic relation between the elements' attributes as well as the references. The correspondence rules have to consider the different kinds of the relation between the artifacts. For example I distinguish between two cases of relation:

- The inherited correspondence, which means that the subclasses of a superclass are correspondent with all the classes, which their superclass is corresponding with. In this case, the correspondence rules can be defined briefly by declaring the rule between the superclasses and indicating that this rule can be inherited.

- The non-inherited correspondence means that the subclasses of a class may have correspondences, which differ from the correspondences of their superclass. In this case the correspondency rules of these subclasses have to be declared separately and clearly.

- The semi-inherited correspondence, which is mixture of the abovementioned correspondences. In this type some of the subclasses inherit the same correspondence, whereas the other not. The correspondence between the Task of AMALTHEA and

The Task of ASCET is semi-inherited correspondence and is explained in details in the section 4.2.3.2.

Moreover, I sort the correspondences rules according to the difficulty of the relation into two kinds:

- The simple correspondence rules, which determine the correlated artifacts and describe correspondences between their simple features.
- The nested correspondence rules, which describe the nested correspondences between the references of the artifacts in addition to the simple correspondences between the attributes.

The abovementioned classification will ease the use of these rules in the following steps, like by legacy models integration.

These rules will be used to keep the consistency between the different views. They will be also used by the proposed integration strategy to identify the correlated objects that belong to different correlates legacy models. Therefore, I distinguish also between two types of the relation between attributes /references of the correlated metaclasses. The first type can identify whether two objects are correspondent or not. The symmetry between the first type attributes/ references of two objects reflects that these objects are correspondent and represent common semantic. As a result, these objects have to be linked with each other, even if there are no match of the values of the second type related attributes. I have marked these attributes as identifier attributes. In contrast the second type of the attributes/ references cannot determine whether their objects are correlated or not. To clarify the meaning of identifier attributes I will give an example from Bosch case study. The two corresponding metaclasses called *Task* that belong to different metamodels (AMALTHEA metamodel and ASCET metamodel) have the attributes *name* and *priority*. These attributes are also correspondent with each other. The attribute *name* can be classified as *identifier* attributes. That means if AMALTHEA task object and ASCET task object have identical values of the *name* then these objects are correspondent with each other. However that is not the case by *priority* attribute. If these objects have identical values of the *priority* attribute, they can be correspondent or not.

In the following section 4.4.2 I will explain the use of this classification by identifying and linking the correlated artifacts.

4.4.2 S2: Build the correspondences between legacy models

The goal of this step is to find the corresponding elements which ensure the defined correspondence rules and link them together. To achieve this goal the legacy models are traversed sequentially. For each object that may fulfill one of the defined correspondence roles, the strategy will search for its corresponding object in the other legacy model through comparing the values of the identifier attributes/ references defined in the correspondence rules. If the correspondence rules are simple, then it is enough to compare the *identifier* simple features of the objects with each other. If they are identical then these object will be linked and the potential conflicts by the other attributes will be resolved in the next step of the integration strategy.

By the nested correspondence rules we need also to check the correspondences between the *identifier* objects referred by the related objects in order to make decision whether they are correlated or not. If we found that they are correspondent, then we link them as well as the objects that they refer to. This step can be performed separately or integrated with the next step (find and resolve the conflicts between the legacy models), in order to traverse the models only once. By this traversing the correspondences will be added and the conflicts will be solved as it will be explained in the suggested algorithms in section 4.5.

4.4.3 S3: Find and resolve inconsistent cases

This step detects the inconsistent cases and conflicts. Then it resolves them depending on Vitruvius change-based development approach. To achieve this goal the algorithm shown in the figure 4.11 is applied. This algorithm illustrates the implementing of the second and third steps (S2 and S3) of the strategy of integrating legacy models in change-based development. According to this algorithm the artifacts which have no relation with the artifacts of the other model will be ignored whereas the other artifacts that have correspondence with artifacts of the other model are traversed consecutively in order to ensure that they fulfill the defined corresponding rules. The order of traversed artifacts is important. First the root artifacts are checked then their sub elements and so on.

For each artifact the algorithm searches for their corresponding artifacts by comparing the identifier-key attributes that are defined in the last step. If the corresponding objects are found, then they will be linked with this artifact (as it has been illustrated in the last step 4.4.2) and the identical of the non-identifier attributes values will be checked. The potential conflicts of the non-identifier attributes values is the first type of the inconsistency that is detected by this algorithm. This type of the inconsistency can be resolved semi-automatically by one of the two following solution. The first one is by asking the developer to determine the correct value. The second one is by choosing a default synchronization direction between for each pair of the legacy models, which allows to update values of the target attributes according to the values of the source attributes.

The second type of the inconsistency which can also automatically be detected and resolved is the absent of the corresponding artifacts or the nested corresponding artifacts. To solve this type of inconsistency the algorithm assumes that the artifact, which its corresponding artifacts are not found in the related legacy models, has been newly created and thus generates the appropriate atomic changes (create change), in order to resolve this inconsistent case automatically. These atomic changes are saved in a list and trigger the synchronization by Vitruvius after traverse all legacy models. Consequentially, the change-driven model transformations will create the missed corresponding artifacts and link them also with the related artifact.

4.5 Modeling using external tools

For the discussion of RO4 the modeling using external tools, I distinguish between two cases of using Vitruvius. The first benefits of all the features of Vitruvius like View-based

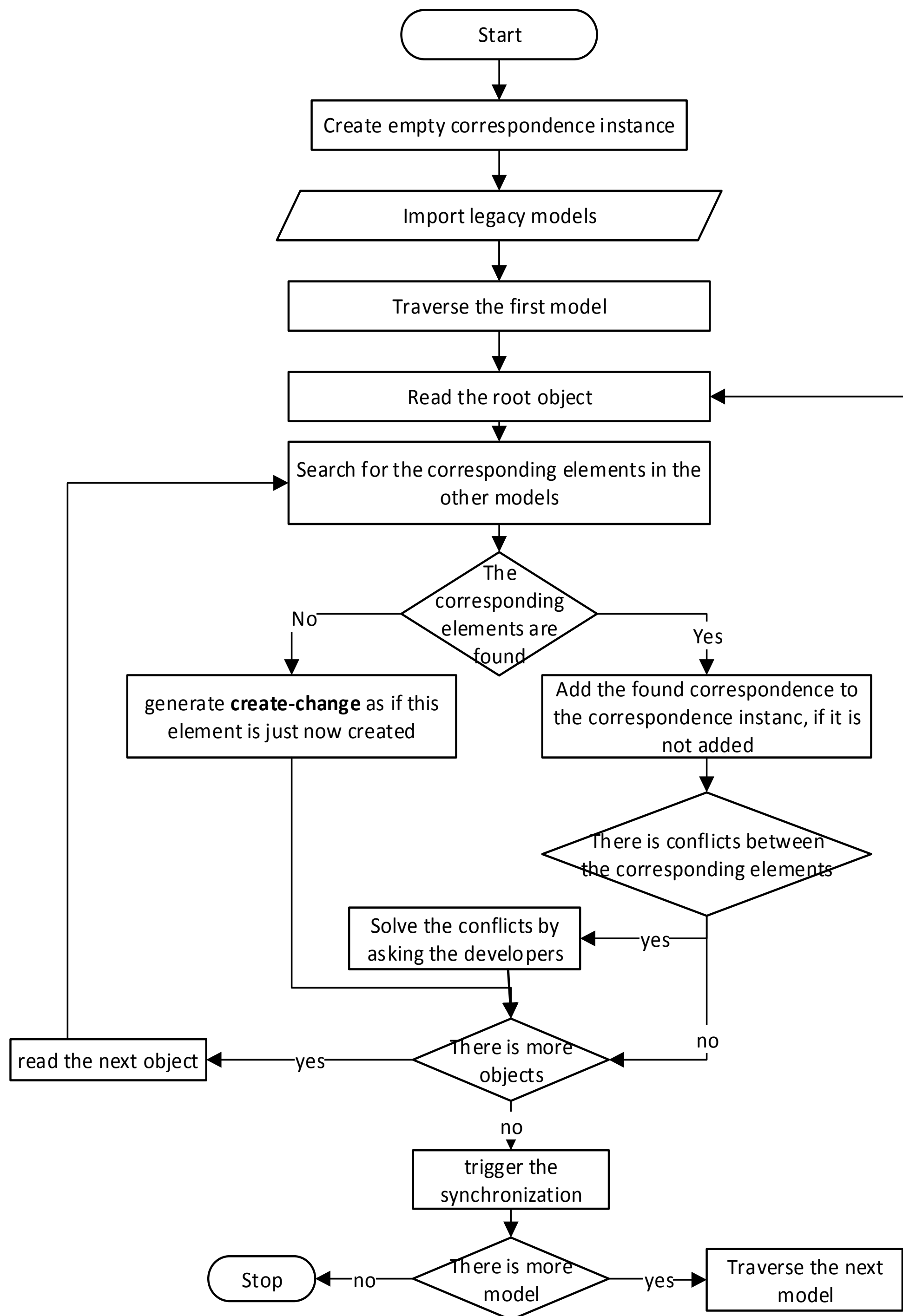

Figure 4.11: Resolve conflicts during the integration of legacy models.

modeling and Consistency preservation. The second concentrates only on using Vitruvius to declare the correspondency rules and resolve potential inconsistencies between the models.

4.5.1 Using Vitruvius as a view-based development approach

Vitruvius is view-and change-based development approach. It depends on recording all the changes during manipulating a model and propagating them to the related views to save the consistency. Applying this approach in the case of using external modeling tools requires implementing bidirectional transmitter, which detects the changes made by the external modeling tools, transforms them to Vitruvius, records the changes applied by Vitruvius during the synchronisation and transform changes applied by Vitruvius to update the models under development in the external tools. Implementing such this transmitter is not always feasible. That is because recording the series of the atomic changes or access the models to update them is not applicable in the case of the closed-source modeling tools (like ETAS ASCET embedded control development systems used in this case study). Therefore, I suggest the following scenario to the collaborative usage of Vitruvius as view-based approach.

In this scenario the developers integrate the automotive legacy models developed in the external tools for the first time using the integration strategy explained in the last chapter 4.4. If there is no legacy models, then only the first step ***S1*** of the strategy is performed in order to create the metarepository including the metamodels and the correspondences between them. Then the developers check out their copies and develop them iteratively. In each iteration the models are modified in the external tools. During the modification of the models the atomic changes have to be either recorded (if it is possible) or calculated using the differences between last two versions. Then the atomic changes are imported to Vitruvius and trigger the synchronisation. As a result, Vitruvius will update the related views and resolve potential conflicts semi-automatically. To reflect the changes executed by the synchronization, the consistent resulting models are exported to the external tools again and new iteration begins.

If the automotive models are not EMF-based format as the case of ASCET models, which is in AMD format(ASCET Modeling Data), then we need an adapter, which can transform the models to/from EMF-based format by implementing the suitable model-based transformations. As a result, importing/exporting the model from/to the external models will be done using this adapter. Moreover, the atomic changes can be then detected from the EMF-based models using comparison tools like EMF compare [12, 51], which calculates the changes between two versions of models. According to the above, we can summarize the use of Vitruvius as a view-based development approach using external modeling tools as shown in Flowchart 4.12.

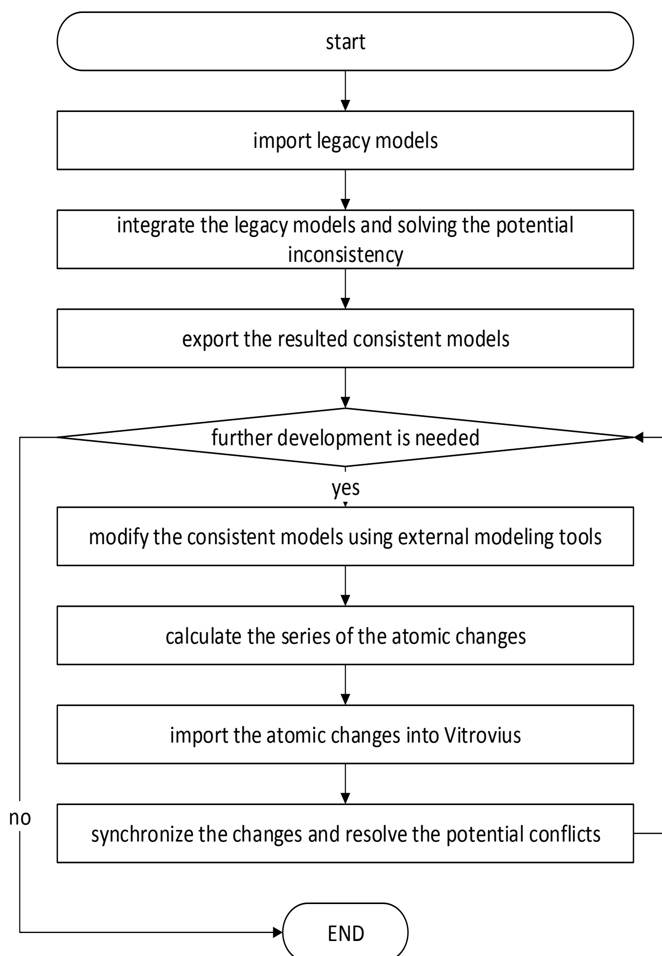

Figure 4.12: Vitruvius development process using external tools.

4.5.2 Using Vitruvius to ensure the consistency between legacy models

In this usage, Vitruvius is used from time to time to check the consistency and resolve potential inconsistent cases between the automotive models. In this case, it is not important to record or calculate the atomic changes. The most recent models are imported to Vitruvius and integrated there. As a result, the abovementioned integration strategy (described in section 2.3.3) will check the consistency automatically and resolve potential inconsistency semi-automatically.

If needed, the resulting consistent models can be further developed using the external models. Then the modified models can be once again imported to Vitruvius to check and ensure the consistency.

The following figure 4.13 illustrates the process of using Vitruvius to preserve the inconsistency between the models created and modified using external tools.

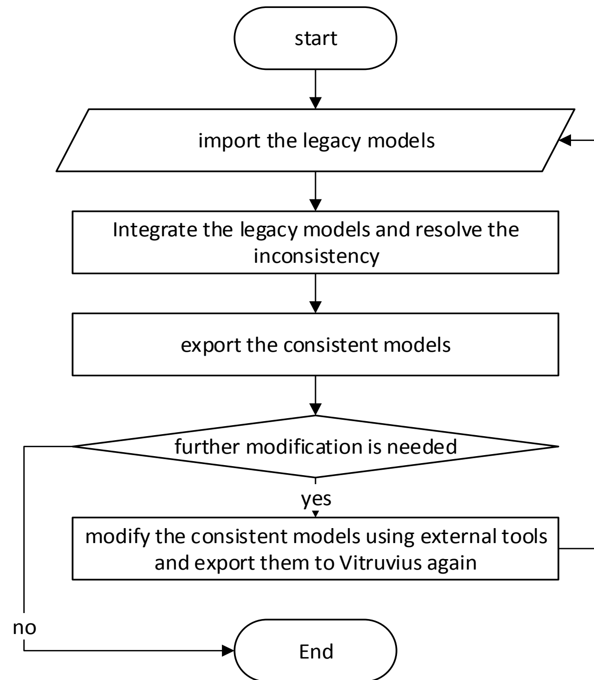

Figure 4.13: Using Vitruvius to resolve the inconsistency of the models, which are created and modified using external tools

5 Evaluation

The evaluation of my case study can be divided into the steps defined in my research objective in section 4.1. In this chapter, I will explain the results and conclusions reached in this research. Consequently, I will answer the research question from section 3.5.

5.1 Evaluation of the correspondences rules expression using MIR language

This section discusses the result of applying the first research objective ***RO1*** which is divided into two steps: detecting the correspondence rules and evaluating the expression of them using MIR language. This discussion will answer the first Research question ***RQ1***.

The following sub-section 5.1.1 introduces the main correspondences between the metamodels of the case study. The subsection 5.1.2 explains how the found correspondences are evaluated. The results of MIR evaluation are summarized in the section 5.1.3

5.1.1 The main correspondences between the case study metamodels

This section explains the main founded correspondences between the AMALTHEA, ASCET and SysML, which are related to the case study explained in the section 3.3.

5.1.1.1 The correspondence between AMALTHEA and ASCET

The implementation of the software components defined in AMALTHEA is executed in ASCET. Therefore, there are semantic overlaps between these models. The AMALTHEA model is divided into the following sub-models: components model, configuration model, constraints model event model, hardware model, mapping model, operating system model, property constraints model, stimuli model and software model. The main correspondences between AMALTHEA and ASCET are between the main model of ASCET on the one hand and both of the software model and component model of AMALTHEA on the other hand. In the following some examples about these correspondences:

Examples:

- **Component and AscetModule**: The components model consists of the components of the system in addition to the definition of the system architecture using System class, which contains several components and the connection-instances between them (see figure 8.5 in appendix).

The AMALTHEA components (shown in figure 5.1) describe the software architecture and are implemented in ASCET using AscetModule type(figure 8.7). AscetModule is one of the component types in ASCET, which describes a number of processes that can be activated by the operating system.

Hence, the first correspondence is between Component and AscetModule metaclass.

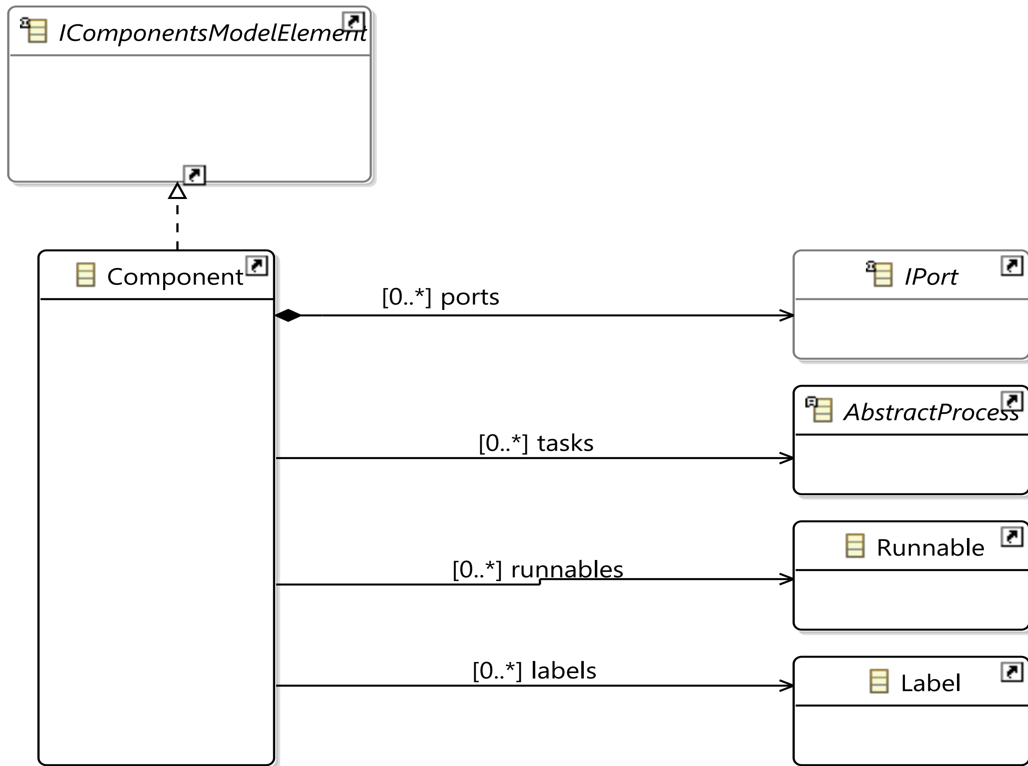

Figure 5.1: The component type in AMALTHEA

- **Runnable-Method:** The component in AMALTHEA defines the runnables. The actual implementation of these runnables are described using the processes of the AscetModule. These Processes are methods called from an operating system task and have neither arguments nor return values. As a result, the runnables are correspondent with the methods achieving the aforementioned conditions.
- **Label-Message:** The components in AMALTHEA exchange the data using data elements named labels (see figure5.2). Each runnable consists of different elements (shown in figure 8.6), one of which is LabelAccess that enables the access to labels based on selected access permission (read or write). Similarly, the data is exchanged between the components in ASCET using Messages type. These Message objects are accessed according to the value of their boolean attributes (read and write) shown in figure 5.3.

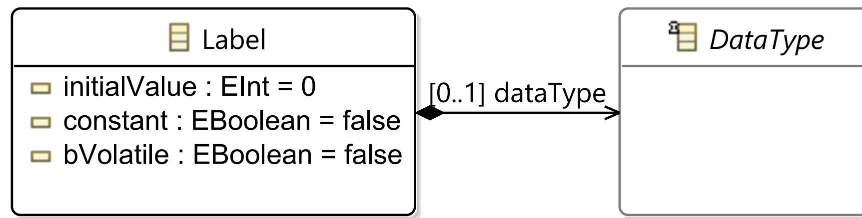

Figure 5.2: The Label type in AMALTHEA

- **Label-different types of data elements:** The data element Label in AMALTHEA can be used as a parameter, a temporarily existing variable or a constant value (see figure 5.2). In ASCET there are different types of data elements like Parameter, Constant, SystemConstant, Variable, Message, Input, Output and Argument (see figure 5.3). These data types can be mapped to the Label type in AMALTHEA and vice versa according to their use.

  The problem of mapping one element to many elements (this problem has been defined in section 4.2.3.2) appears again in this correspondence. But in this case the value of the attribute constant of label can reduce the number of the possible corresponding elements. In other words, if the value of the boolean attribute is false then the label can be mapped to one of the following classes: Variable, Message, Argument, Input and Output (Argument, Input and Output types can be also ignored because they are not used in the scenario of this case study). On the other hand if the value of this attribute is true label will be mapped to one of the following classes: Parameter, Constant and SystemConstant. In both of the cases the problem of mapping one element to many elements is still existed.

- **Task-Task:** The software model in AMALTHEA defines the tasks, which call one or various sequences of the runnables, according to some constraints. In a similar way different types of tasks are defined in ASCET. Each of them defines a set of processes and is corresponding to the task type in AMALTHEA. This correspondence has been explained in details in sections 4.2.3.2, 4.2.3.1.

- **ISR-Task:** Similar to the previous correspondence the InterruptTask is corresponding with the type ISR (Interrupt Service Routines) in AMALTHEA, which is for interrupt processes (see the example in section 4.2.3.2 and the related figure 4.5).

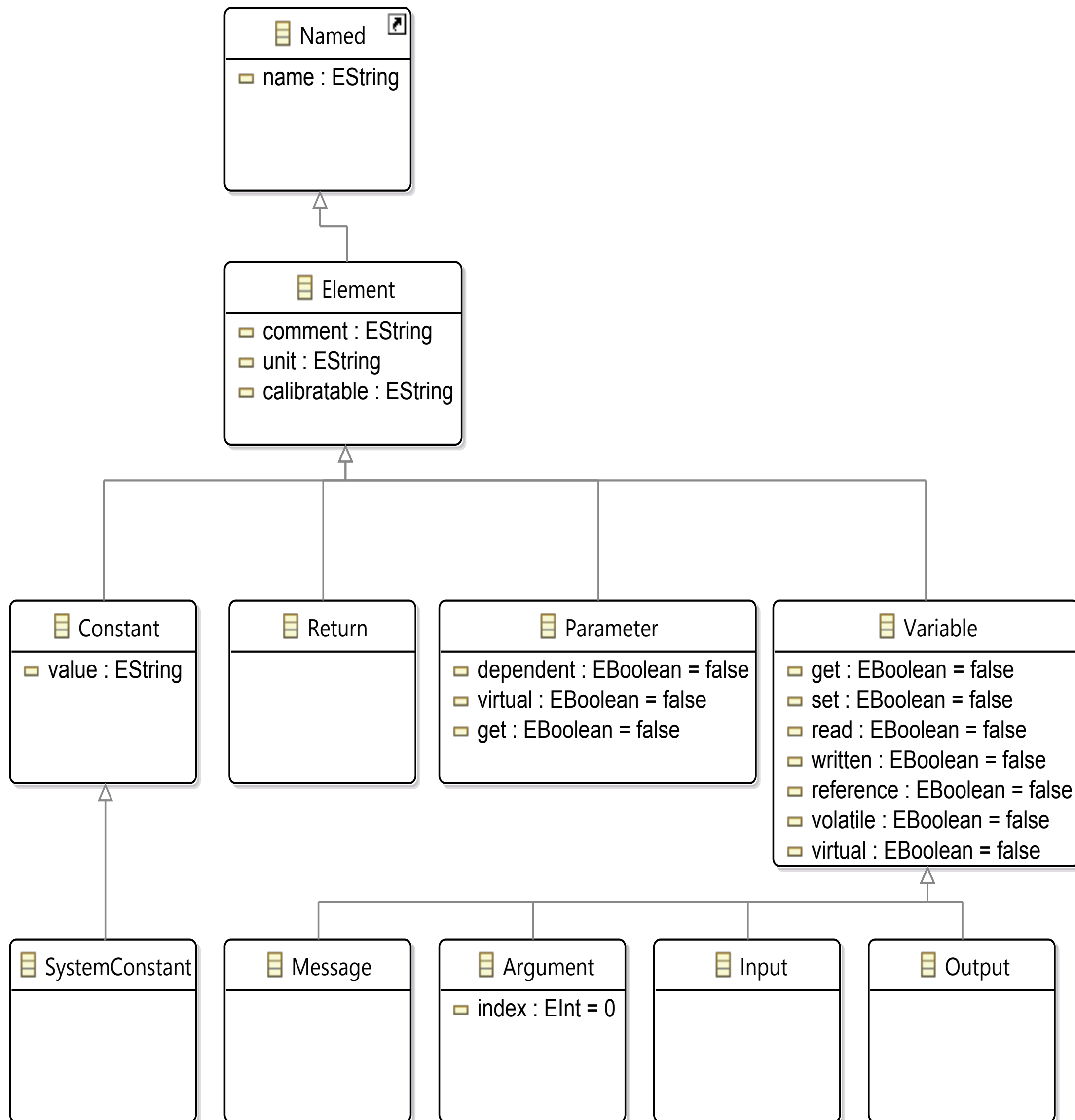

Figure 5.3: ASCET data types

Table 5.1: Examples of the correspondences between AMALTHEA and ASCET metamodels

AMALTHEA metaclasses	ASCET metaclasses	OCL constraints	Correlated attributes/ references	
			AMALTHEA	ASCET
Component	**AscetModule**		name	Name
			Runnable[] runnables	Method[] methods*
			Labels[] labels	Element[] elements**
Task	**Task, InitTask, SoftwareTask, TimeTableTask**		name	Name
			priority	Priority
			called runnables****	Method[] processes***
ISR	**InterruptTask**		name	Name
			priority	Priority
			called runnables****	Method[] processes***
Runnable	**Method**	**Context** Method **Inv**: self .ret. oclIsUndefined () **Inv**: self -> collect(arguments)-> size()=0		
Label	**Parameter, Constant, Systemconstant**	**Context** Label **Inv**: self.constant=true	name	Name
			Datatype	Type
Label	**Variable, Message, Argument, Input, Output**	**Context** Label **Inv**: self.constant=false	name	Name
			Datatype	type
BaseTypeDefinition	**Log**	**Context** BaseTypeDefinition **Inv**: self.size=1		
BaseTypeDefinition	**Udisc, Sdisc**	**Context** BaseTypeDefinition **Inv**: Set{8,16,32}->includes{ self.size}		
BaseTypeDefinition	**Cont**	**Context** BaseTypeDefinition **Inv**: Set{32,64}-> includes{ self.size }		
Array	**ArrayType**		numberElements	Size
			Datatype	basicType

* Additional sub-constraint: **Context** AscetModule **Inv**: self -> collect(elements)-> oclIsTypeOf(Message)
** Additional sub-constraints: **Context** AscetModule **Inv**: self -> collect(methods) -> ret. oclIsUndefined ()
Inv: self -> collect(methods) -> collect(arguments)-> size()=0
*** Additional sub-constraints: **Context** Task **Inv**: self -> collect(processes) -> ret. oclIsUndefined ()
Inv: self -> collect(processes) -> collect(arguments)-> size()=0
**** All runnables that can be called by this task (see the example illustrated in section 4.2.3.1)

5.1.1.2 The correspondences between SysML and ASCET

The software structure of the automotive system can be defined in ASCET using the block diagram. The blocks represent the system components, which can be ASCET module or ASCET classes. The same semantics can be described in SysML either with the BDD or IBD diagram.

For example in our case study the block ControlAlgorithm defined in SysML(shown in figure 3.3) can be represented in ASCET as AscetModule named also ControlAlgorithm (shown in figure 3.7).

Based on this example, I will give in the following more example about the correspondences between the ASCET and SysML model.

- **Block-AscetModule** ASCET module represents a number of processes that can be activated by the operating system. This component cannot be used as a subcomponent within other components. Therefore it is corresponding with the Block metaclass in SysML, when this block is not encapsulated in other Block. In other words the value of the boolean property isEncapsulated has to be false. For example the ASCET module ControlAlgorithm is corresponding with the Block object which has the same name. This correspondence is explained in details in section 4.2.3.4 too.

- **Block-AscetClass** Classes in ASCET are like object-oriented classes. Their functionality is described by methods which have oft arguments as input and return type as output. For example, in Bosch case study both of the PIDT1 and Limiter components of the ASCET module ControlAlgorithm are instances of AscetClass type (see figure 3.7). These classes are corresponding with the parts of the Block contrlAlgorithm. In other words, the AscetClass metaclass has correspondence with the Block metaclass when this block is part of other block.

- **FlowPort-Message** As it is illustrated in section 3.3.2 the data can be followed in and out the ASCET module using Message type. Therefore the correspondence between the AscetMdule type and Block type will be nested to include the correspondences between the messages (type of Message) of AscetModule and the flow ports, which belong to the Block (type of FlowPort). The direction of the flow port will be determined according to the value of boolean attributes *read* and *written* defined in Message type.

- **FlowPort-Argument** Unlike ASCET module the ASCET class receives the data using the arguments of its method. This means that the arguments of a ASCET class (type of Argument) will have sub-correspondence with flow ports of Block (type of FlowPort) when the direction of flow port is *in*.

- **FlowPort-Return** Similar to the previous correspondence the return of the methods defined in ASCET class (type of Return) will have a correspondences with the out flow ports of the block correlated with their ASCET class.

Table 5.2: Examples of the correspondences between SysML and ASCET

SysML metaclasses	ASCET metaclasses	OCL constraints	Correlated attributes/ references	
			SysML	**ASCET**
Block	**AscetModule**	Context Block Inv: self. isEncapsulated= false	Name	name
			flowPorts**	Element[] elements*
FlowPort	**Message**	Context FlowPort Inv: Block.allInstances()-> select(x|x.base_Class-> collect(ownedports)-> includes(self.base_Port)). isEncapsulated= false	Base_Port. name	name
			Direction=0	written=false read=true
			Direction=1	written= true read=false
			Direction=2	written=true read=true
Block	**AscetClass**	Context Block Inv: self. isEncapsulated= true	Name	name
			flowPorts**	Element[] elements*
FlowPort	**Argument**	Context FlowPort Inv: Block.allInstances()-> select(x|x.base_Class-> collect(ownedports)-> includes(self.base_Port)). isEncapsulated= true Inv: self.direction=0	Base_Port. name	name
FlowPort	**Return**	Context FlowPart Inv: Block.allInstances()-> select(x|x-> collect(flowparts)-> (base_Port) -> includes(self.base_Port)). isEncapsulated= true Inv: self.direction=1	Base_Port. name	name

***Additional sub-constraint: Context AscetModule Inv: self -> collect(elements)-> oclIsTypeOf(Message)**
**** The flow ports that belong to a block can be collected as following:**
Context Block
Inv: FlowPort.allInstances()-> select(x |self.base_Class-> collect(ownedPort)-> includes(x.(base_Port)) .

5.1.2 The evaluation and the feedback of the found correspondence rules

The found correspondences rules are summarized in both of the table 5.1 and the table 5.2, which illustrate the mapping between the different types in addition to the pre-/ post-conditions that defined using OCL. After that these correspondences are discussed and evaluated with Bosch's experts. Some of the found correspondences are ignored, like the correspondences between Argument, Input and Output types from ASCET on one hand and Label type from AMALTHEA on the other hand because they are not used in the scenario of this case study. Moreover other correspondences are ignored in the implementation like the correspondences between the data types. Because the model types in ASCET are transformed to the implementation (data) types through describing the implementation process. As a result, there is no concise information about the implementation type during the design, like the size of type in memory.

5.1.3 The result of MIR evaluation

The first research objective ***RO1*** aims to evaluate the features of MIR language by Bosch case study. As mentioned in section 2.4 MIR language is still under development and its editor only supports defining the declarative mapping. Therefore, only the mapping blocks (which is described in section 2.4.1) are evaluated. Besides that, the implementation needed for the automatic generation of the required model transformations are also not finished. Hence, the evaluation does not include the evaluation of the correctness of the generated model based transformations. In other words, the evaluation includes only the ability of expression the mapping between the abovementioned correspondences 5.1.1 in readable and high level of abstraction.

The expression of the mapping in MIR suffers from some problems discussed in the section 4.2.3. This section will summarize them according to their importance. The most important problem is the inability of expressing the correspondences whereas the less important one is related to its effect on the quality. I mean with the quality the level of clarity and abstraction of rules, which can be measured by the number of the MIR rules needed to express one correspondence.

Depending on this aspect the found problems can be divided into two types:

- Problems by defining the correspondences. For example the correspondences which have either no direct access on a metaclass or require referring to any order of objects cannot be declared using the current version of MIR. These problems meet 11 correspondences (the former problem meets 5 and the last meets 6) and applying the suggested solutions will enable MIR to define one MIR correspondence rule for each correspondence.

  Other example is the problem of mapping one metaclass to many metaclasses. Although the correspondences can be declared in MIR but the defined MIR correspondence rules lack information that determines to each one of these multiple metaclasses the single corresponding metaclass has to be mapped.

  Applying the suggested solution (described in 4.2.3.2) will allow defining the rules correctly and can raise also the level of the quality through preventing the repetition

Table 5.3: Summary of MIR problems

The MIR Problem	Correspondences	Number of needed rules	
		with this problem	after applying suggested solution
No direct access on a metaclass	5	0 (cannot be declared)	5 rules
One-to-many mapping	19	7 using pre-condition	7+6 rules
Referring to any order of objects	6	0 (cannot be declared)	6 rules
Duplicate writing similar rules	3	31 similar rules	3 rules
By all the problem	23	1 rule	17 rules

of similar rules by the case of inherited correspondences. This problem meets 19 of 23 correspondences of this case study.

However, 7 of these correspondences can be declared in the current version of MIR by adding some precondition. For example the Block in SysML is corresponding with both of the ASCET module and ASCET class, but these correspondences can be distinguished according to the value of the attribute isEncabsulated. (see table 5.2). The rest 12 correspondences can be defined by 6 MIR rules using the suggested solution.

- Problem by the quality of the MIR correspondence rules. For instance the problem of repeating similar rules. In this case study it is required to repeat 31 MIR rules in order to declare only 3 correspondences. Solving this problem will allow to explain these 3 correspondences using only 3 MIR rules instead of 31 one. Solving this type of problem is less important than the former type (problems by defining the correspondences) because the correspondences can be defined but with low quality.

The table 5.3 summarizes the abovementioned problems and indicates the number of the correspondences that have this problem in the second column *correspondences*. Additionally, the table compares between the number of the needed MIR rules to declare these correspondences before and after applying the suggested solution, regardless of the other problems and without applying their suggested solutions. The last row of table clarifies that from the total 23 correspondences only one of them can be defined with this MIR version. Furthermore, 17 MIR rules will be needed to define the rest 22 correspondences after using the suggested solution.

5.2 Applying the extended Vitruvius development process

In my case study the models used by Bosch are modeled using different tools: SysML modeling tool like Enterprise Architect and papyrus (I depended on papyrus tool and its metamodel, because it is open source), AMALTHEA tool platform and ETAS ASCET embedded control development systems (V6.3). Vitruvius offers no editor for these models. Moreover, I have more than one legacy model, which have to be integrated in Vitruvius platform. Therefore, this Version of Vitruvius cannot be applied to this case study. Consequently, I can answer the second research question ***RQ2: "How should the development***

process of Vitruvius be evolved in order to support the automotive systems development?" that the extended development process has to support the integration of the legacy models and the development using external tools. The main steps of the extended development process have been defined in section 4.3 and shown in figure 4.3. According to these steps I build the metarepository of the related metamodels (SysML, ASCET and AMALTHEA) and implement the proposed strategy for integrating the legacy models (this strategy is introduced in section 4.4). The section 5.3 will describe the results of this implementation.

Furthermore, the section 5.4 will discuss the development process using external tools based on this implemented strategy. Then the section 5.5 will compare the results of the consistency preservation using the extended development process of Vitruvius with the traditional consistency preservation used by Bosch.

5.3 Support and integration of the legacy models by Vitruvius

The suggested strategy described in the section 4.4 has been implemented and evaluated with Bosch case study. For this purpose, I have applied the first step of the strategy and created VSUM metamodel. Performing this step **S1** does not differ from the old version of Vitruvius. According to the correspondence rules defined in the last section I have expressed and implemented the most important of them using Xtend (see section 5.3.1). That is because MIR language is still under development and cannot generate the required transformation rules. Then the next steps **S2** and **S3** are implemented (see section 5.3.2) in order to build the correspondences between the imported models and resolve the inconsistency between them. The results of applying this strategy will be summarized later in the section 5.3.3.

5.3.1 The mapping of the correspondences

For the mapping I have use Xtend programming language, whose syntax is slimmer than java. It is integrated very well with Eclipse IDE and is suitable to express model transformation using large available library. Moreover, it generates the java code automatically. Therefore It is used by the implementation.

However, mapping using Xtend compared to MIR language needs more efforts. Because more code will be written to express the model-based transformation between the metamodels in the two directions (from the first metamodel to the second metamodel and visa versa).

Therefore,I implemented the model-based transformations only between AMALTHEA and ASCET as a first step. These transformations represent the main rules between the software components in AMALTHEA and their implementation component called ASCET Module in ASCET, in addition to the correspondence between the runnables and the correlated processes in ASCET on one hand and the labels and the messages on the other hand.

Hence, the metaclasses included in the transformations are:

- AMALTHEA: ComponentsModel, Component, SwModel, Runnable and Label.

- ASCET: AscetModule, Method and Message.

The evaluation: The implemented transformations are tested using different defined test cases. In each one a test model file is created and then new elements are created, added to the model and manipulated. When the defined transformations are correct then the corresponding elements will be effected correctly after triggering the synchronization using Vitruvius platform. The results are evaluated using assertion functions, which show that the transformation are correctly implemented.

5.3.2 The integration of AMALTHEA and ASCET models

To evaluate the integration strategy I implement the algorithm shown in figure 4.11 in order to build the correspondences between the imported legacy model, check of the consistency and resolve potential conflicts.

Evaluation: In order to evaluate different cases, the legacy model integration algorithm is evaluated by the following test case in addition to the Bosch case study (the result of Bosch case study is explained in the section 5.4). The reason is that this case study has more inconsistent cases. In this test case three inconsistent legacy models are imported. The first model is ASCET model (.xadom file). The second one is AMALTHEA software model (sw-amxmi file), which saves the all software elements like tasks, runnable and labels. The third one is the AMALTHEA components model (.components-amxmi file), which distributes the defined software elements to specific components. The figure 5.4 shows the three legacy models with its elements. The AMALTHEA components model has two components, one of them is correlated with the ASCET module named *comp1*. However not all the methods defined in ASCET module (M1,M2) are declared as runnables in the correlated AMALTHEA component *comp1*. Furthermore the runnable R1 in the AMALTHEA component *comp1* is not implemented in the correlated ASCET module.

According to the legacy model integration algorithm these emf-based inconsistent models are consequently traversed and then the correspondences are added and saved in Vitruvius platform. Adding the correspondences between the object depends on the defined correspondences (the identifier attributes, pre-condition, etc.). For example when the object is instance of Runnable then the algorithm searches for an object of Method type in ASCET, which have the same name of runnable, no arguments and no return type. When it is found, then it links the two objects (runnable object and the found method, which represents a process and defines the implementation of this runnables) and adds them to the created correspondence instance. An example of the case study is that the algorithm links the runnable *RM* of the component *comp1* with the method *RM* of the correlated ASCET module *comp1*. As a result, the correspondences between these models are added. Moreover, the conflicts are resolved through creating and adding the required objects to the models. To achieve this purpose the algorithm supposes that the objects, whose corresponding elements are not existed, have just been established. According to this suppose it creates then the related change (create-change) and supplies this change

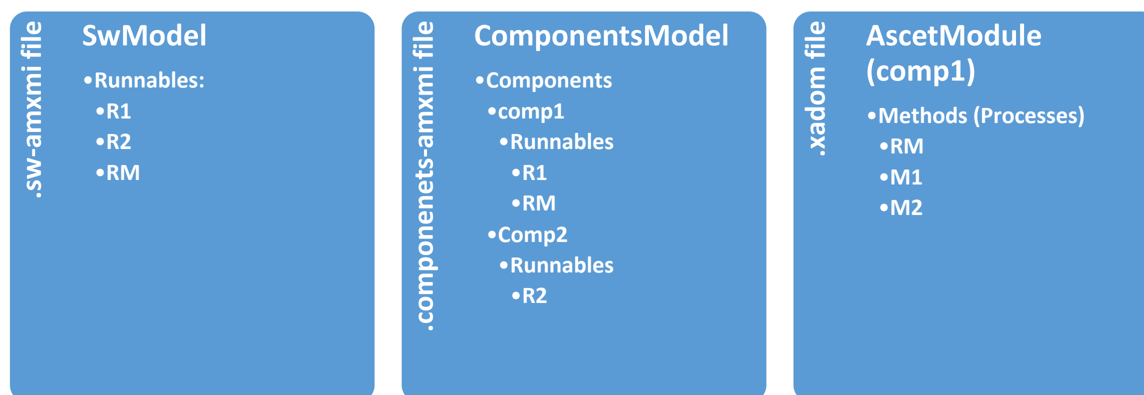

Figure 5.4: The inconsistent legacy models of the test case before the integration into Vitruvius platform (the round rectangles represent the files of the models and the bulleted list represent the model's elements).

to Vitruvius platform. As a result Vitruvius will create the corresponding objects after triggering the synchronization.

For example, if the algorithm finds no method in ASCET model, which can be considered as implementation for an existing runnable in AMALTHEA like the case of runnable *R1* of the component *comp1*, then it creates create-change for this runnable and as a result Vitruvius will create a suitable method for this runnable and add it to the ASCET model. Creating this method will depend on the defined model-based transformation. In my example Vitruvius will create a Method named *R1* and has neither arguments nor return type. Then it will add this Method to correlate ASCET module *comp1*). As a result of applying this algorithm, the missed corresponding elements will be created and added to the models as it is shown in following figure 5.5. The objects, which are written with yellow color, are created by Vitruvius to resolve the found inconsistency.

5.3.3 Results

Applying the abovementioned strategy can help by integrating the legacy models in Vitruvius under the following condition:

- The imported models have to be valid. For example, AMALTHEA metamodel allows that the Task object have multiple Stimulus (see figure 8.1 in appendix), but at most one of these Stimulus can be type of the subclass periodic (the class stimulus and its subclasses are shown in figure 8.2.1.2 in appendix). If the model has a Task object with more than one Periodic object, then this model is invalid. Integrating this invalid model will cause a problem when the invalid Task object is transformed to the PeriodicTask type in ASCET, because it will be not clear which one of the multiple Periodic objects has the correct information about the time attributes (recurrence and offset).

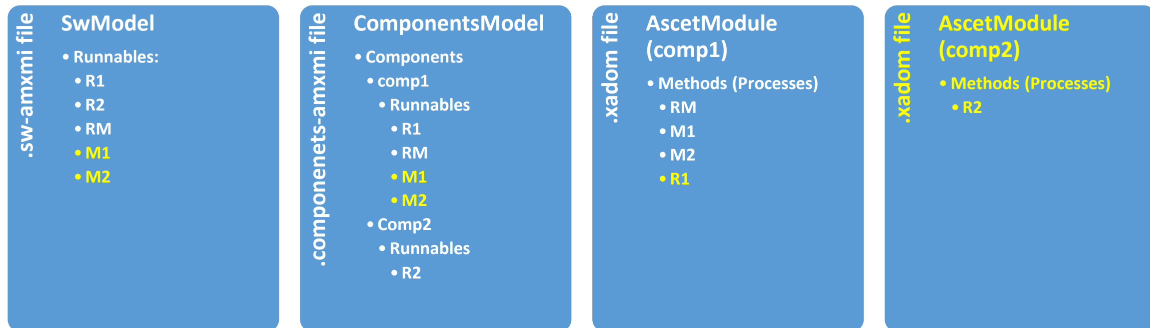

Figure 5.5: The resulting consistent test case models after resolving the inconsistency by applying suggested synchronization algorithm.(the round rectangles represent model files and the bulleted list represent the model's elements. Elements created by Vitruvius to resolve the inconsistency are colored with yellow).

Other example is that the runnables' names as well as the components' names have to be unique, otherwise neither the mapping nor the linking with the related element can be successfully executed. The developers of Bosch comply usually with these restriction. However, these constraints can be defined and checked again in Vitruvius but this implementation does not implement such restrictions. It supposes that the imported models is valid.

- The integration of the legacy models cannot be always executed automatically. In some cases the developers have to make decision how the inconsistency have to be resolved. For example when one element is corresponding with many elements.

- If the inconsistency found because of missing the related elements, then this strategy solves the inconsistency always by creating these missed elements whereas the inconsistency can be also resolved by deleting the element, whose corresponding elements are not exist. However, the developer is informed about the changes and can delete this element and the new added corresponding elements.

- The implementing of this strategy, which integrates more than two models in the same time, may lead to recreate of existing objects. For example, if the algorithm integrates the models of AMALTHEA, ASCET and SysML in the same time and there is an object of AscetModule named "ControlAlgorithm", then the strategy will search for an AMALTHEA Component object named "ControlAlgorithm" and SysML Block object with the same name too. If only one of these objects is existed, then the create-change will be created in order to resolve the inconsistency and create the missed object. The synchronisation by Vitruvius will propagate this change and create both of the two corresponding objects (Component object in AMALTHEA and Block object in SysML). Consequently, one of these object will be recreated.

This Problem can be solved either by the incremental integrating or by defining the suitable constraints and responses. The first solution requires repeating the algorithm several times, as long as each pair of the models should be integrating with each other. By using this solution by the case study the integration algorithm will integrate AMALTHEA and ASCET models first, then the resulted ASCET model will be integrated with SysML model.

The second solution defines the appropriate invariants and responses, which prevent the recreation of the existing object. In the last example the defined invariant will prevent creating two objects with the same name and ignore the later created object as a response of violation of this invariant. This response is not optimal by the further development, because the developer can unintentionally create two Component objects (for example) with the same name and this violation of invariant can be solved by changing the name of the recent created object, and not by ignoring it. Therefore, the optimal response of this invariant is to ask the developer in order to decide whether to ignore the recent object or to change its name, if this constraints is violated.

- The automatic resolution of the inconsistency will create the missed corresponding objects with the shared information only. This means that the updated legacy models have to be reviewed after the integration in order to supply and add the additional needed information. For example when Vitruvius creates a Method object, then it supplies this object only with the name and the additional information about it has to be added in ASCET. Similarly, when an AMALTHEA Task object is created by Vitruvius, then all its runnables will be created and saved in one call sequence whereas AMALTHEA allows defining different call sequences and selecting one of them during the runtime according to other data (see section 4.2.3.1). So, the developer can update the created Task object in AMALTHEA platform and divide the calling of the created runnables into different call sequences if it is needed.

- Checking the consistency between two elements and linking them is based on the identifier attributes. The conflicts of the values of the non-identifier attributes can not be automatically resolved. The developer will be asked to determine the correct value every time.

According to the previous discussion I can answer the third research question ***RQ3****"How can the scenario of "legacy" models be supported by Vitruvius?"* as the following. Vitruvius can support the development of the legacy models by applying the suggested integration algorithm shown in figure 4.11 under the abovementioned conditions. Building the correspondences and checking the consistency can be done automatically. Resolving the potential inconsistency are done semi-automatically. The resulted consistent models have to be reviewed in order to add the additional needed information.

5.4 Vitruvius development process using external tools

In this section, I will answer the following research question ***RQ4*** : *"How can Vitruvius be used when the used models are developed and updated using other tools?"*. In my case study the models are developed and updated using external modeling tools. Therefore, I have suggested two possible scenarios to use Vitruvius (see section 4.5).

The first one uses Vitruvius continuously as view-based development approach. This scenario will benefit from the approach of asynchronous collaborative software development (see 2.3.1). In this approach the developers will check out their copies, manipulate them using external tools, calculate the made changes in EMF format through recording them or comparing between two versions of models, check in the changes in Vitruvius and resolve potential conflicts.

The advantages of this scenario is supporting view-based development and ensuring the consistency along the development process.

The requirements of this usage is the detection of the changes made during the development in external tools. This can be done through adding a listener to records these changes (if it is possible, like by open source EMF-based modeling tools) or by calculating the changes by comparing between two versions of models, which can be done using EMF-compare techniques [12] when the models are based on XMI (XML Metadata Interchange [37]). If the models are not EMF-based then an adapter which transform the models to/ from EMF-based model will be also needed.

The second scenario uses Vitruvius in selective points for the purpose of checking the consistency and resolving the potential conflicts. In this scenario the legacy models are imported to Vitruvius. Then they are integrated using the integration strategy explained in the section 4.4. During the integration the strategy checks the consistency and resolves the prospective inconsistent cases semi-automatically. Finally, the models may be exported to the external tools again either to supply some additional needed information to the elements created automatically by Vitruvius or to develop them further. Moreover, further consistency verification may be done by integrating the new versions of the models again in Vitruvius.

This scenario offers only semi-automatic consistency preservation. In the best case, the consistency will be checked by the extended Vitruvius prototype only two times (the first to detect and resolve potential inconsistency and then to insure that the models are still consistent after the further development of the performed changes).

According to the previous I can summarize and answer the fourth research question***RQ4*** as following: Vitruvius can be used in one of the two aforementioned scenarios. Choosing the preferred scenario is based on the desire of developers to use Vitruvius on an ongoing basis as view-based development or only occasionally to ensure the consistency. In the two cases the inconsistent cases will be detected and resolved efficiently in the design stage. As a result, the burden and time needed by the used approach will be reduced (as it will be declared in the section 5.5).

Unfortunately, the test case could not determine which of the aforementioned usage is more efficient because the case study is not complicated compared to the real project and both AMALTHEA and ASCET models are already formed. Therefore, the second scenario of using Vitruvius is adopted for the purpose of checking the consistency of these models

before generating and integrating the code. The result of the case study is discussed in the following subsection (section 5.4).

The result of using Vitruvius to ensure the consistency of the case study As aforementioned in the last section Vitruvius approach is used to preserve the consistency between the models of the case study (defined in section 3.3).

The consistency is verified using Vitruvius two times. In the first time the legacy models shown in the figure 5.6 are integrated in Vitruvius.

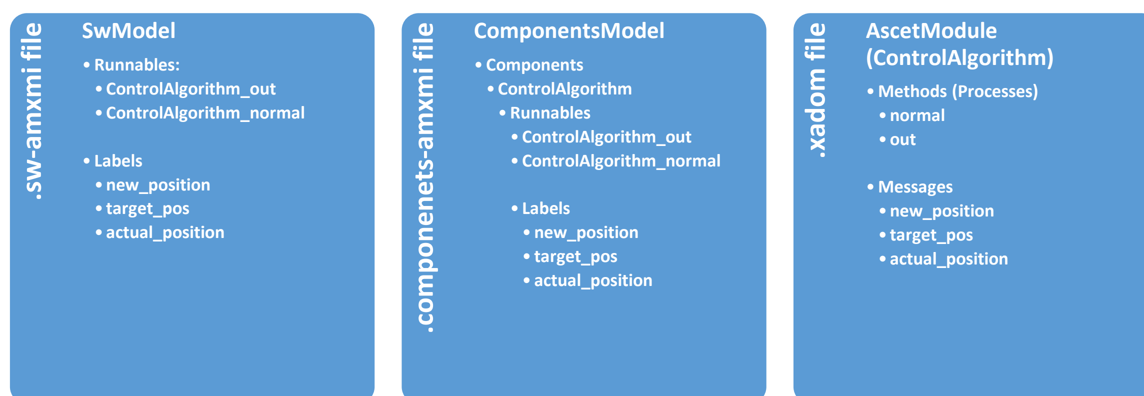

Figure 5.6: The inconsistent case study legacy models. (the round rectangles represent model files and the bulleted list represent the model's elements.)

According to the defined correspondences rules the inconsistency is detected. The inconsistent cases are:

- There is two Runnable objects (ControlAlgorithm_normal, ControlAlgorithm_out) in AMALTHEA model, whereas there are no processes that implement their functionality in ASCET.
- The two Method objects (normal, out) represent processes in ASCET but there are no runnables in AMALTHEA model, which represent these execution units.

As a consequence, the integration algorithm resolves the inconsistency by creating two Runnable objects with the same name of the existed Method objects (normal, out) on one hand and creating two Method objects with the same name of the defined Runnable objects (ControlAlgorithm_normal, ControlAlgorithm_out) on the other hand (the figure 5.7 shows the models after resolving the conflicts and creating the new objects that written with yellow color).

The integration strategy export the updated models and informs the developer about these updates. Then the developer reviews them and finds that the reason of the inconsistency is the different naming convention. Consequently, the developer deletes the new created runnables and methods and modifies the names of the runnables to (normal, out) as it is shown in figure 5.8. After that the consistency is verified for the second time. The

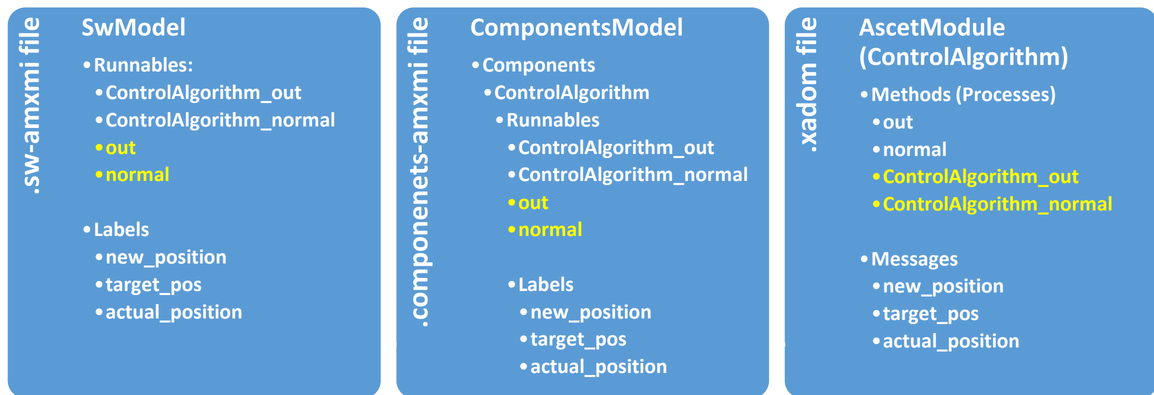

Figure 5.7: Case study models after the first integration in Vitruvius platform. (the round rectangles represent model files and the bulleted list represent the model's elements. Elements created by Vitruvius to resolve the inconsistency are colored with yellow)

models are integrated again in Vitruvius. As a result, the consistency of the case study models are confirmed.

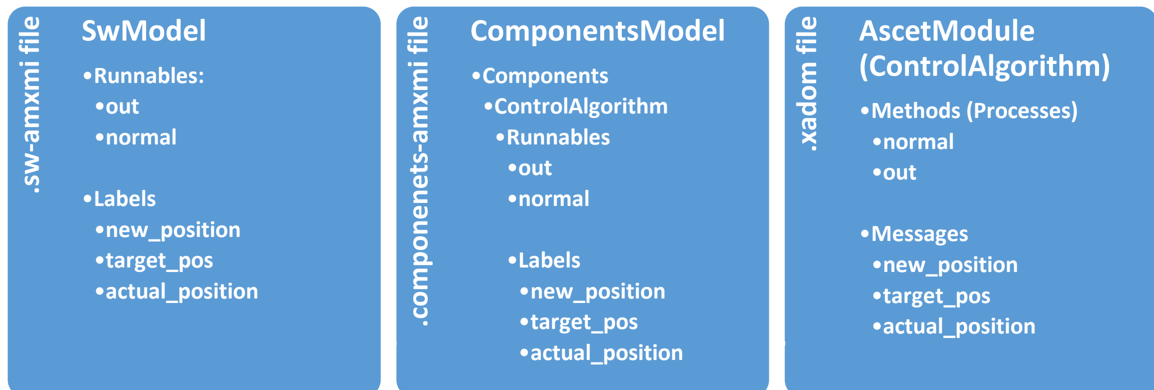

Figure 5.8: The consistent case study models after the second integration in Vitruvius platform (the round rectangles represent model files and the bulleted list represent the model's elements.)

5.5 Evaluate the consistency preservation by Vitruvius in practice

The developers of automotive system aim to develop consistent automotive models with the lowest cost. Therefore, they seek detecting and resolving the potential inconsistency

as early as possible. This section summarizes the consistency preservation on the basis of applying Vitruvius extended development process, which supports the integration of the legacy models on the one hand and the development using external modeling tools on the other hand. This section answers also the following questions:

- ***RQ5:*** Can the effort needed to detect and resolve the potential inconsistent be reduced by applying Vitruvius approach?

- ***RQ6:*** How far can the automatic consistency preservation between automotive systems be achieved?

The traditional consistency preservation approach detects some of the inconsistent cases for the first time in the implementation stage using the compiler and linker. These cases causes bugs when the before generated codes are integrated and compiled. Resolving the inconsistency in this late stage needs a great effort. That is because the compilation process takes a long time (several hours) and it has to be restarted after correcting errors. Comparing to this method, I can answer the fifth research question ***RQ5*** with yes. Using the extended Vitruvius development process will reduce the burden and the time needed to check the consistency and resolve potential inconsistent cases. That is mainly because this process will save the additional time and manual effort needed by traditional approach for generating the codes, integrating these codes with each other, translating the integrated code, debugging the code and fixing the errors.

In contrast to the traditional approach, Vitruvius avoids also the drift and erosion between the code and the design on one hand and detects all the inconsistent cases on the other hand, as long as they are defined by the corresponding rules.

Besides that, the case study shows that the detected inconsistent cases cannot be always automatically resolved. Thus, I answer the last research question***RQ6*** with the following, it is not possible to perform the fully automatic consistency preservation because the conflicts are semi-automatically resolved. There are several cases, which need asking the developer about the correct solution. For example, the case of the difference between the values of the related non-identifier attributes. In this case the inconsistency will be resolved according to the choice of the developer, who determines the correct value.

Other example is the case of the one-to-many mapping. In this case the developer cannot always select one of the multiple types as a default to perform the mapping, otherwise the developer will be asked every time to choose the correct. For instance, the corresponding object of an AMALTHEA BaseTypeDefinition object, whose size attribute has the value *32 bit*, can be mapped to one of the following ASCET types:

- Udisc (unsigned discrete, which is mapped by the implementation to the type *uint* with 8, 16 or 32 bit).

- Sdisc (signed discrete, which is mapped by implementation to the type *sint* with 8, 16 or 32 bit).

- Cont (continuous, which is mapped by implementation to real32 or real 64)

The developer in this case cannot choose one of these different types as a default when he defines the correspondence rules. Because selecting the suitable type depends mainly on the ASCET element and its use. Therefore, the best solution in this case will be asking the developer.

Besides that, the automatically resolution of the inconsistency has to be also evaluated and reviewed by the developer in order to add the additional needed information to the elements created automatically on one hand and to correct the resolutions that are not performed properly on the other hand. For example naming convention conflicts is not resolved properly (see the concrete example in section 5.4).

6 Related work

The related work will be presented here in two sections. First, I will present related consistency preservation approaches in section 6.1. The traditional approach which depends on the documents interchange will be explained in the section 6.1.1. Other model-based approaches addressing the consistency problem are discussed in the section 6.1.2. A related work to integrate legacy models in change-based development is discussed in section 6.2.

6.1 Consistency preservation in automotive system development

During the automotive development process, different heterogeneous models are developed separately by different project partners using several modeling tools. These models are often correlated and share the same semantic. Therefore, the project partners have to keep them consistent. To achieve this goal one of the following two approaches or mixture between them can be applied. The first one is document-centric approach, which keeps the consistency through exchanging the information and documentation in form of documents along the development process. The second one is model-based approach which reduce the manual effort using the model-based transformation for the synchronization. The following subsections present some works based on these approaches.

6.1.1 Consistency preservation using document-based approach

In document-based approach the information is exchanged using documents along the development process to ensure the consistency. This approach suffers, that some of these documents are written by hand. Moreover, the stored information is reused in this approach through copy-and-paste techniques.

The exchanged documents store related information. Consequently, each change in an individual artifact will require update in multiple documents and there is no efficient mechanism to determine the affected documents. In the following some works, that are based on this approach.

consistency preservation based on MSR standard Manufacturer Supplier Relationship MSR is a consortium of a car manufacturers and suppliers, which supports common developments between car manufacturers and their electronic system suppliers by enabling process synchronization and proper information exchange based on XML. AE division of Bosch

applies technology based on this standard in addition to XML standard [49] [1] in order to achieve a consistent information handling throughout the entire software development process [52]. According to this technology the information is stored uniformly using only consistent formats for definition and exchange all relevant information. These formats are provided from MSR standard which defines for this purpose several XML Document Type Definition (DTD) based on common set of definition and practices. DTDs allow storing information from different models to reduce the redundancy of information and ease the consistency preservation. The uniformly defined information is saved only once in a data bank to enable the exchange of the information between project partners in any point of the development process.

This approach suffers from the manual production of the MSR documents in addition to the manual synchronization, which can lead to inconsistency. Moreover, this approach does not support efficient mechanism to check the consistency, which is checked often in too late stage using the compiler and linker. Implementing checks based on MSR files is also possible and can detect more inconsistencies than compiler and linker tools but it requires more effort. Resolving the conflicts is also performed manually and needs a great effort since it is performed in late stage.

Consistency checking by applying ISO 26262 standard ISO 26262 [34, 40] specifies a functional safety life-cycle for automotive systems comprised of electrical, electronic and software components.

Born et al. [9] suggests using safety analysis of ISO 26262 which requires efficient and consistent product development. Therefore, they represent the exchanged information (even if it is a document) as a models in order to separate the artifacts from their external representation. Moreover they prevent the redundancy of the information through storing them in a single source and allow generating or importing the documents from this resource.

Model-based representation of the exchanged documents eases the tracing of the distributed information. As a result semi-automatic consistency checking can be performed.

In my work the consistency can be fully automatically checked. Moreover the potential conflicts can be resolved semi-automatically, whereas they are still resolved manually by Born et al.

6.1.2 Consistency preservation depending on model-based approach

The model-based approach aims to tackle the difficulty of the document-centric approach through using the model-to-model transformation techniques to offer automatically development (e.g. driving one model from the other) or automatically synchronization of the heterogeneous models.

Model Synchronization based on triple graph grammars Triple graph grammars (TGGs) are a formalism for declarative description of correspondence relationships between two

[1] http://www.msr-wg.de/

types of models. This concept was introduced in [43] in order to generate the required bidirectional model transformations using a declarative transformation specification. Contrast to MIR approach, TGGs do not support the rules that delete elements.

TGGs have been employed in several domains. Some examples are, generating the bidirectional transformation between SDL and UML models during integrating them within the fujaba tool suite [16], preserving the consistency of models from the domain of chemical engineering [5] and integrating of SysML models with Modelica simulation models [25].

Another example from the automotive systems domain, TGGs is employed for the purpose of the synchronization between system engineering models in SysML and software engineering models in AUTOSAR [19].

The synchronization in this approach can be only done between two models. If there are more than two models, then chains of transformations should be built to connect them. However, that is not the case in our approach where the synchronization between several models are applied using VSUM, which propagates the changes to all models that are affected by these changes.

The changes are propagated by the approach of Giese in two modes, the synchronisation mode and transformation mode. In the first mode the consistency in the target model will be checked after executing the transformation. When the changes violate the consistency constraints in the target model, they will be ignored. That is not the case in our approach where the appropriate actions can be defined during the declarative description of the consistency rules. Consequently, these actions will be performed when the related constraints are violated. Similar to our approach, one model can be driven from other legacy model.

By the case of two legacy models in Giese approach, the changes in the source model are propagated to the target one and by conflicts the elements of the target model will be updated. Moreover, some changes could be ignored when they lead to the inconsistency in the target model. In my work the bidirectional propagation of the changes will be available. By the conflicts, the developer will be asked to determine which elements have to be updated; in other words, the resolving of conflicts is not related of the synchronization direction.

Consistency management using macromodels Salay et al. [41] define the relationships between the models in dependence on macromodel concept. In this concept a formal method is used for the specification of model relationships (mapping and constraints). Depending on macromodels any change made to a model will be checked formally using techniques such as logic inference rules and constraint satisfaction in order to determine the existence of inconsistency.

This concept is applied in vehicle control system development. The relationship between two types of flow diagrams in a functional architecture model (Functional Architecture Diagram (FAD) and Component Diagram (CpD)) is defined using macromodel. Then some inconsistencies between these models can be detected using macromodels, and may repaired automatically, through formal expressions of model relationships.

This work shows positive impacts on inter model consistency during evolution of the system but it does not support the inconsistency detection and conflicts reservation between the legacy models.

Developing consistent models using model transformations Selim et al.[44] apply model transformations to migrate from the legacy models of General Motors (GM) company, which are built using custom-built and domain-specific modeling language, to the standardized AUTOSAR models. These model transformations are validated by a case study using the MDWorkbench tool, the Atlas Transformation Language and the metamodel Coverage Checker tool.

Model-to-model transformations have been used also by Sindico et al. [45] in order to generate Simulink models from SysML models and vice versa.

Similar work [46] is also performed to transform Simulink models to UML composite structure and activity models based on Atlas Transformation Language ATL.

The abovementioned approaches are limited to a specific combination of two models or languages, and generate the related model of a legacy one based on the model transformation into one direction. Moreover they have no mechanism to ensure the consistency between two legacy models like the case of our approach.

Consistency preservation depending on Seamless Model-based Development approach This approach is based on the integration of the different models covering all phases of system development from system requirements to system design and verification in order to ease the information exchange between them and to ensure the consistency between them [11]. However, applying this approach suffers from enormous political and technical barriers, which are illustrated from Broy et al [11].

Macher et al. [30] depend on seamless combination of the heterogeneous tools to improve the continuity of information interchange of architectural designs from system development level to software development level through adopting bidirectional tool-bridge. The application of this approach implements tool-bridge, which supports exporting software architectures designed in SysML to Matlab/Simulink tool on one hand and importing the software module implemented in Matlab/Simulink on the other hand. Moreover model-driven software engineering tools is used for adding more details about the software architecture, software modules and the correspondences between them. This eases the tractability between software architecture and software modules and ensures the consistency along the development process.

Another work of Macher [31] generates the configuration of automotive real-time operating systems OSEK (like allocation to a CPU respectively to a task) using control system information in SysML (such as control strategies) in order to ensure the consistency and the correctness required by automotive safety standards (such as ISO 26262). The work also enables the update of the configuration information saved in OSEK Implementation Language (OIL) file in addition to the possibility of importing the information from OIL files. For that purpose a bidirectional tool-bridge between model-driven systems engineering tools and software engineering for automotive real-time operating systems (RTOS) tools is established.

The approach of Macher is similar to Vitruvius approach, where both of them support the model-based development of consistent models and extraction of the corresponding models from a legacy one using model transformations.

However, the synchronisation mechanism of Vitruvius approach is easier, when more than two models are developed (see the section 2.2.2). Furthermore, the extension of Vitruvius enables the integration of more than one legacy model.

6.2 Integration of the legacy models in the change-based development

The authors of [29] present two integration strategies that could be applied to either a model or code in order to integrate them in a change-based development. The first one is **r**econstructive **i**ntegration **s**trategy (**RIS**), which traverse the elements of the legacy models and suppose that these elements have been just now created. Consequently, the suitable create-changes for each element will be generated and propagated by the change-based development tool (Vitruvius prototype is used in the case study). As a result, the legacy model will be recreated and its correspondence model will be also established. The implemented version of the reconstructive strategy allows the integration of PCM models into Vitruvius. During the integration, related Java code and a correspondence model are created.

The second strategy **l**inking **i**ntegration **s**trategy (**LIS**) relies heavily on the reverse engineering tool SoMoX. It allows the integration of source code (It is implemented for the Java source code) and its component models (PCM model in the case study) , generated using reverse engineering tools SoMoX. The correspondences between the source code and component model are added depending on the information resulted by the reverse engineering Process.

The approach of Giese et al. [19] enables also the integration of one legacy model in change-based development. According to this approach the elements, which match the defined rules, will be transformed and their correspondences will be stored. As a result the related model can be generated from a legacy one.

The two abovementioned strategies integrate only one legacy model in the change-based development. However, this work represents strategy to integrate multiple legacy models in the change-based development environment.

7 Conclusion

In this chapter I will summarize the results and draw the conclusions of my work in the next section 7.1. Furthermore, I will describe the following steps to extend the work in the future work section 7.2.

7.1 Summary

This work has extended Vitruvius approach in order to improve the automotive system development and keep the consistency between the different automotive models that are separately developed.

First, the work evaluates the declarative expression of the automotive correspondences rules using MIR through defining the most important correspondences between the automotive metamodels used in the case study (SysML, AMALTHEA and ASCET). The problems and restrictions found in the current version of MIR are documented and illustrated by examples. Furthermore, the thesis suggests some solutions and ideas to improve MIR and enable using it for the declarative expression of the correspondences in high level of abstraction.

Second, the thesis presents and implements a strategy to integrate multiple legacy models in Vitruvius. The consistency of these models are checked automatically during the integration. Besides that, the strategy enables mechanism to solve the potential conflicts semi-automatically based on Vitruvius approach. This integration strategy is important to enable applying Vitruvius concept, when the models are developed using modeling tools that are not supported from Vitruvius prototype.

Third, the work presents two scenarios for adopting the Vitruvius approach when the heterogeneous models are developed using external modeling tools. The first one uses Vitruvius continuously and benefits from the synchronization mechanism of Vitruvius to keep the models consistent along the development process. The second one uses Vitruvius from time to time to integrate the legacy models and benefit of the automatic consistency check and semi-automatic resolving of conflicts.

As a result, this work offers more efficient mechanism for the consistency preservation during the automotive system development. In this mechanism the inconsistent cases can be detected earlier comparing to the traditional concept, which detects only some of them after the modeling stage when the generated codes are integrated and compiled. Consequently, the time and effort needed for detecting and resolving the potential inconsistent cases will be significantly reduced. Furthermore, the proposed mechanism avoids also the drift and erosion between the models and the implementation of the system, because the detected conflicts are resolved semi-automatically in the design stage.

7.2 Future work

The future work of this research will build on the obtained results in three ways.

First, we will develop the MIR language in order to cope with the problems encountered and the restrictions and be able to generate the bidirectional transformations needed. This allows us to develop the semi-automatic repairing methodology of identified inconsistencies. Second, we will test the integration of the legacy models and evaluate it with other test cases in order to assess the robustness. Third, we will implement the second strategy of using Vitruvius in the automotive development system and evaluate it by use cases. This strategy enables using Vitruvius along the development process and is based on recording or calculating the changes done by the external tools. The EMF-based models (like AMALTHEA and SysML) are in EMF format and allow the calculation of changes using EMF-comparison techniques. The other models (like ASCET model, which is in XML-AMD format) need building an adapter, which can transform the model from/to the EMF-based model. (In this work we use an AMD-to-ADOM adapter in order to convert the ASCET model (XML-AMD Format) to ADOM (EMF format). However, this adapter has to be improved to be able to transform the model from ADOM to XML-AMD format too). These changes have to be transferred to Vitruvius in order to update the models and synchronize them with each other.

Bibliography

[1] Colin Atkinson, Dietmar Stoll, and Philipp Bostan. “Orthographic software modeling: a practical approach to view-based development”. In: *Evaluation of Novel Approaches to Software Engineering*. Springer, 2010, pp. 206–219.

[2] Colin Atkinson et al. “Modeling components and component-based systems in kobra”. In: *The Common Component Modeling Example*. Springer, 2008, pp. 54–84.

[3] AUTOSAR. *worldwide development partnership of car manufacturers, suppliers and other companies from the electronics, semiconductor and software industry*. `http://www.autosar.org/`. 2012.

[4] K Beck, M Beedle, A Bennekum, et al. *Manifesto for Agile Software Development. Agile Alliance (2001)*. 2012.

[5] Simon M Becker et al. “A graph-based algorithm for consistency maintenance in incremental and interactive integration tools”. In: *Software & Systems Modeling* 6.3 (2007), pp. 287–315.

[6] Steffen Becker, Heiko Koziolek, and Ralf Reussner. “The Palladio component model for model-driven performance prediction”. In: *Journal of Systems and Software* 82.1 (2009), pp. 3–22.

[7] Lorenzo Bettini. *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt Publishing Ltd, 2013. ISBN: 9781782160304.

[8] Jean Bézivin. “On the unification power of models”. In: *Software & Systems Modeling* 4.2 (2005), pp. 171–188.

[9] Marc Born, John Favaro, and Olaf Kath. “Application of ISO DIS 26262 in practice”. In: *Proceedings of the 1st Workshop on Critical Automotive applications: Robustness and Safety*. ACM. 2010, pp. 3–6.

[10] Christopher Brink and Jan Jatzkowski. *AMALTHEA (ITEA2 - 09013) - White Paper*. Tech. rep. University of Paderborn, Germany, 2013.

[11] Manfred Broy et al. “Seamless model-based development: From isolated tools to integrated model engineering environments”. In: *Proceedings of the IEEE* 98.4 (2010), pp. 526–545.

[12] Cédric Brun and Alfonso Pierantonio. “Model differences in the eclipse modeling framework”. In: *UPGRADE, The European Journal for the Informatics Professional* 9.2 (2008), pp. 29–34.

[13] Erik Burger. “Flexible Views for Rapid Model-Driven Development”. In: *Proceedings of the 1st Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling*. VAO ’13. Montpellier, France: ACM, 2013, 1:1–1:5. ISBN: 978-1-4503-2070-2.

[14] Erik Burger. "Flexible Views for View-Based Model-Driven Development". In: *Proceedings of the 18th international doctoral symposium on Components and architecture*. WCOP '13. Vancouver, British Columbia,Canada: ACM, 2013, pp. 25–30. ISBN: 978-1-4503-2125-9.

[15] Erik Burger. "Flexible Views for View-based Model-driven Development". PhD thesis. at the department of Informatics, Institute for Program Structures and Data Organization - KIT, Karlsruhe Institute of Technology, July 2014.

[16] Sven Burmester et al. "Tool integration at the meta-model level within the fujaba tool suite". In: *In Proc. of the Workshop on Tool-Integration in System Development (TIS)*. 2003.

[17] Jordi Cabot and Martin Gogolla. "Object constraint language (OCL): a definitive guide". In: *Formal Methods for Model-Driven Engineering*. Springer, 2012, pp. 58–90.

[18] Sanford Friedenthal, Alan Moore, and Rick Steiner. "OMG Systems Modeling Language (OMG SysML) Tutorial". In: *INCOSE International Symposium*. Vol. 18. 1. Wiley Online Library. 2008, pp. 1731–1862.

[19] Holger Giese, Stephan Hildebrandt, and Stefan Neumann. "Model synchronization at work: keeping SysML and AUTOSAR models consistent". In: *Graph transformations and model-driven engineering*. Springer, 2010, pp. 555–579.

[20] Thomas Goldschmidt, Steffen Becker, and Erik Burger. "View-Based Modelling – A Tool-Oriented Analysis". In: *Proceedings of the Modellierung 2012*. Ed. by Elmar J. Sinz and Andy Schürr. Vol. P-201. GI-Edition – Lecture Notes in Informatics (LNI). Bamberg, Mar. 2012.

[21] Etas Group. *ASCET Model-based Software Development*. Flyer. Feb. 2016. URL: `http://www.etas.com/download-center-files/products_ASCET_Software_Products/ASCET_6.4_Flyer_EN.pdf`.

[22] Etas Group. *ASCET V7 Tools for Developing Safe and Efficient Software*. Flyer. Feb. 2016. URL: `http://www.etas.com/download-center-files/products_ASCET_Software_Products/ASCET_V7_Flyer_EN.pdf`.

[23] Etas Group. *ETAS, A global and growing company*. Flyer. 2014. URL: `http://www.etas.com/download-center-files/company/etas_global_growing_company.pdf`.

[24] *Industrial automation systems and integration*. International Organization for Standardization (ISO), 1994.

[25] Thomas Johnson et al. "Integrating models and simulations of continuous dynamics into SysML". In: *Journal of Computing and Information Science in Engineering* 12.1 (2012), p. 011002.

[26] Hermann Kopetz et al. "Automotive software development for a multi-core system-on-a-chip". In: *Software Engineering for Automotive Systems, 2007. ICSE Workshops SEAS'07. Fourth International Workshop on*. IEEE. 2007, pp. 2–2.

[27] 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. Montpellier, France: ACM, 2013, 5:1–5:6. ISBN: 978-1-4503-2070-2.

[28] Max E Kramer et al. "Change-Driven Consistency for Component Code, Architectural Models, and Contracts". In: *Proceedings of the 18th International ACM SIGSOFT Symposium on Component-Based Software Engineering*. ACM. 2015, pp. 21–26.

[29] Sven Leonhardt et al. "Integration of Existing Software Artifacts into a View-and Change-Driven Development Approach". In: *Proceedings of the 2015 Joint MORSE-VAO Workshop on Model-Driven Robot Software Engineering and View-based Software-Engineering*. ACM. 2015, pp. 17–24.

[30] Georg Macher, Eric Armengaud, and Christian Kreiner. "Integration of Heterogeneous Tools to a Seamless Automotive Toolchain". In: *Systems, Software and Services Process Improvement*. Springer, 2015, pp. 51–62.

[31] Georg Macher et al. "Automotive real-time operating systems: a model-based configuration approach". In: *ACM SIGBED Review* 11.4 (2015), pp. 67–72.

[32] Harald Mackamul. *Building an open source, extendible development platform*. Tech. rep. Robert Bosch GmbH, 2011.

[33] Harald Mackamul. "Model Based Open Source for Embedded Multi-Core Systems". In: *EclipseCon Europe 2013*. Ludwigsburg, Germany, Oct. 2013.

[34] Johannes Matheis. *Abstraktionsebenenübergreifende Darstellung von ElektrikElektronik-Architekturen in Kraftfahrzeugen zur Ableitung von Sicherheitszielen nach ISO 26262*. Shaker, 2010.

[35] *MOF 2.5 Core Specification (formal/2015-06-05)*. Object Management Group (OMG), June 2015. URL: %5Curl%7Bhttp://www.omg.org/spec/MOF/2.5%7D.

[36] Siegfried Nolte. *QVT-operational mappings: Modellierung mit der Query views Transformation*. Springer-Verlag, 2009.

[37] XML OMG. *Metadata Interchange (XMI) Specification*. URL: http://www.omg.org/docs/formal/05-05-01.pdf. (accessed October 10, 2005). 2000.

[38] K Venkatesh Prasad, Manfred Broy, and Ingolf Krueger. "Scanning advances in aerospace & automobile software technology". In: *Proceedings of the IEEE* 4.98 (2010), pp. 510–514.

[39] *Recommended Practice for Architectural Description of Software-intensive Systems*. http://www.iso-architecture.org/ieee-1471/. ISO. International Organization for Standardization (ISO), 2011.

[40] *Road vehicles - Functional safety*. http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=43464. ISO.

[41] Rick Salay, Shige Wang, and Vivien Suen. *Managing related models in vehicle control software development*. Springer, 2012.

[42] Tariq Samad and Thomas Parisini. “Systems of systems”. In: *The Impact of Control Technology* (2011), pp. 175–183.

[43] Andy Schürr. “Specification of graph translators with triple graph grammars”. In: *Graph-Theoretic Concepts in Computer Science*. Springer. 1995, pp. 151–163.

[44] Gehan MK Selim et al. “Model transformations for migrating legacy deployment models in the automotive industry”. In: *Software & Systems Modeling* 14.1 (2015), pp. 365–381.

[45] Andrea Sindico, Marco Di Natale, and Gianpiero Panci. “Integrating SysML with Simulink using Open-source Model Transformations.” In: *SIMULTECH*. 2011, pp. 45–56.

[46] Carl-Johan Sjöstedt et al. “Mapping Simulink to UML in the design of embedded systems: Investigating scenarios and transformations”. In: *OMER4 Workshop: 4th Workshop on Object-oriented Modeling of Embedded Real-Time Systems*. 2007.

[47] *Allgemeine Modelltheorie*. Springer-Verlag, Wien, 1973. ISBN: 978-3-7091-8327-4.

[48] *Systems and software engineering - Architecture description*. http://www.iso-architecture.org/ieee-1471/. ISOIECIEEE 42010, 2010-2011.

[49] Marek Szwejczewski, Fred Lemke, and Keith Goffin. “Manufacturer-supplier relationships: An empirical study of German manufacturing companies”. In: *International Journal of Operations & Production Management* 25.9 (2005), pp. 875–897.

[50] Martin Törngren et al. “Model-based development of automotive embedded systems”. In: *Automotive Embedded Systems Handbook* (2008).

[51] Antoine Toulmé and Intalio Inc. “Presentation of EMF compare utility”. In: *Eclipse Modeling Symposium*. 2006, pp. 1–8.

[52] Bernhard Weichel and Martin Herrmann. *A backbone in automotive software development based on XML and ASAM/MSR*. Tech. rep. SAE Technical Paper, 2004.

[53] Carsten Wolff et al. “Automotive software development with AMALTHEA”. In: *Practice and Perspectives* (2015), p. 432.

8 Appendix

8.1 Abbreviations

AE	Automotive Electronics
ASCET	Advanced Simulation and Control Engineering Tool
ASCET-MD	ASCET Modeling and Design
ASCET-MIP	ASCCET MATLAB Integration Package.
ASCET-RP	ASCET Rapid Prototyping
ASCET-SE	ASCET Software Engineering
ATL	Atlas Transformation Language
AUTOSAR	AUTomotive Open System ARchitecture
BDD	Block Definition Diagram
CpD	Component Diagram
DSL	domain-specific language
DTD	Document Type Definition
ECU	Electronic Control Unit
ESDL	Embedded Software Description Language
FAD	Functional Architecture Diagram
IBD	Internal Block Diagram
ISO	International Organization for Standardization
JML	Java Modeling Language
LIS	Linking Integration Strategy
MDD	Model-Driven Development
MDE	Model Driven Engineering
MIR	Mappings, Invariants, and Responses
MSR	Manufacturer Supplier Relationship
OIL	OSEK Implementation Language
OSM	Orthographic Software Modeling
PCM	Palladio Components Model
RIS	Reconstructive Integration Strategy
RTE	Real-Time Environment
RTOS	Real-Time Operating Systems
SoS	System of Systems
SUM	Single Underlying Model
SysML	Systems Modelling Language
TGGs	Triple graph grammars
UML	Unified Modeling Language
Vitruvius	VIew-cenTRic engineering Using a VIrtual Underlying Single model
VSUM	Virtual Single Underlying Model
XML	Extensible Markup Language

8.2 AMALTHEA models

8.2.1 Software model

The AMALTHEA software model is central accessible through the SWModel element. The namespace for the model is "http://www.amalthea.itea2.org/model/1.1.0/sw".

8.2.1.1 Task

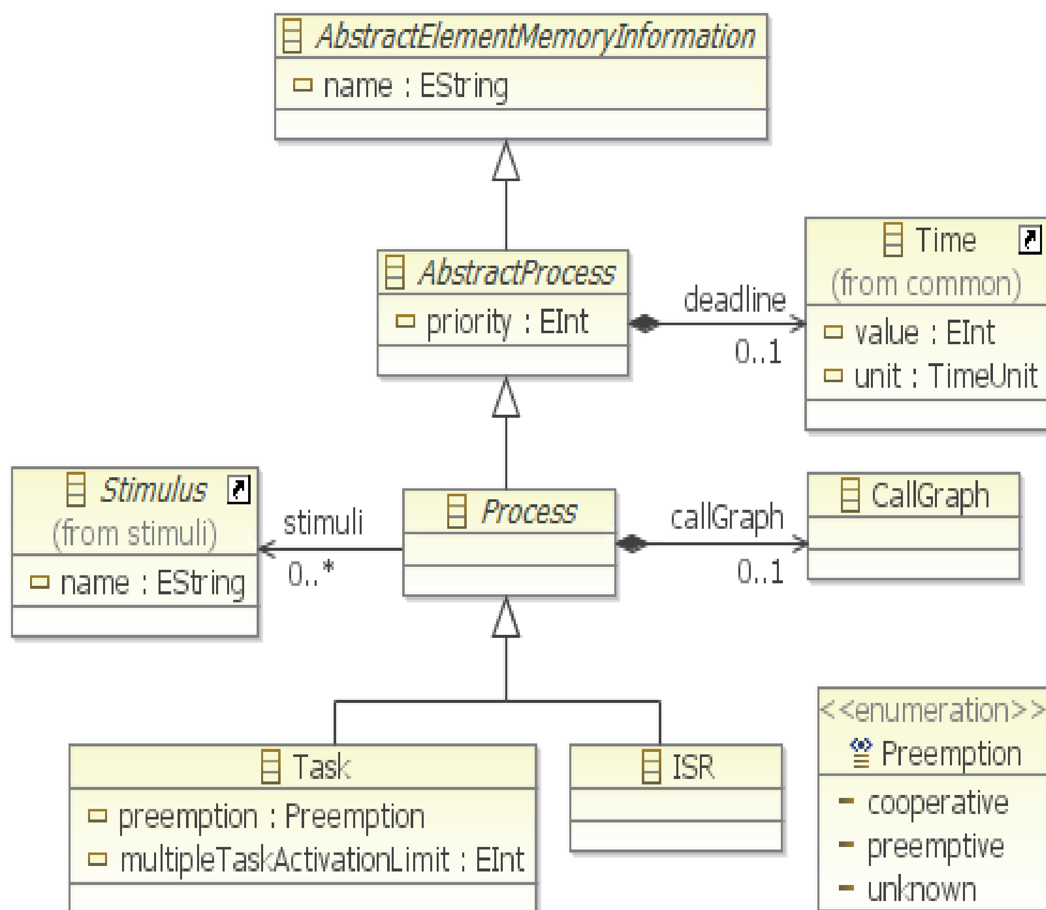

Figure 8.1: Metamodel excerpt for Task, ISR and Stimulus

8.2.1.2 Stimuli model

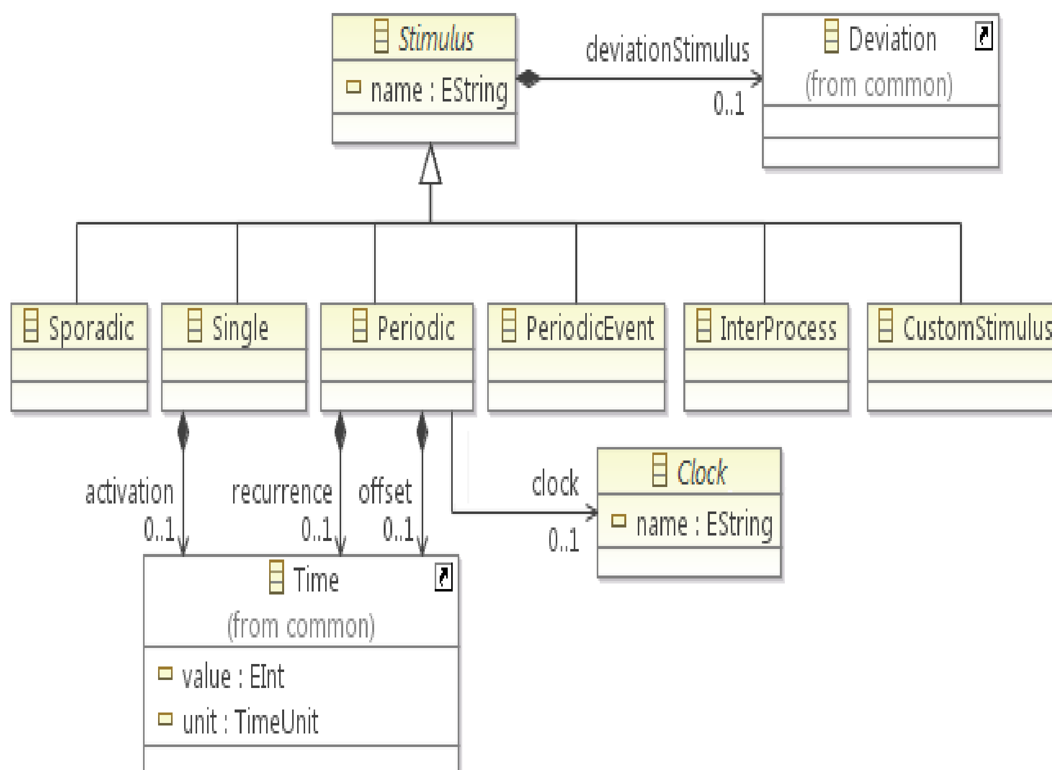

Figure 8.2: Metamodel excerpt for Stimulus and Periodic

8.2.1.3 Data types

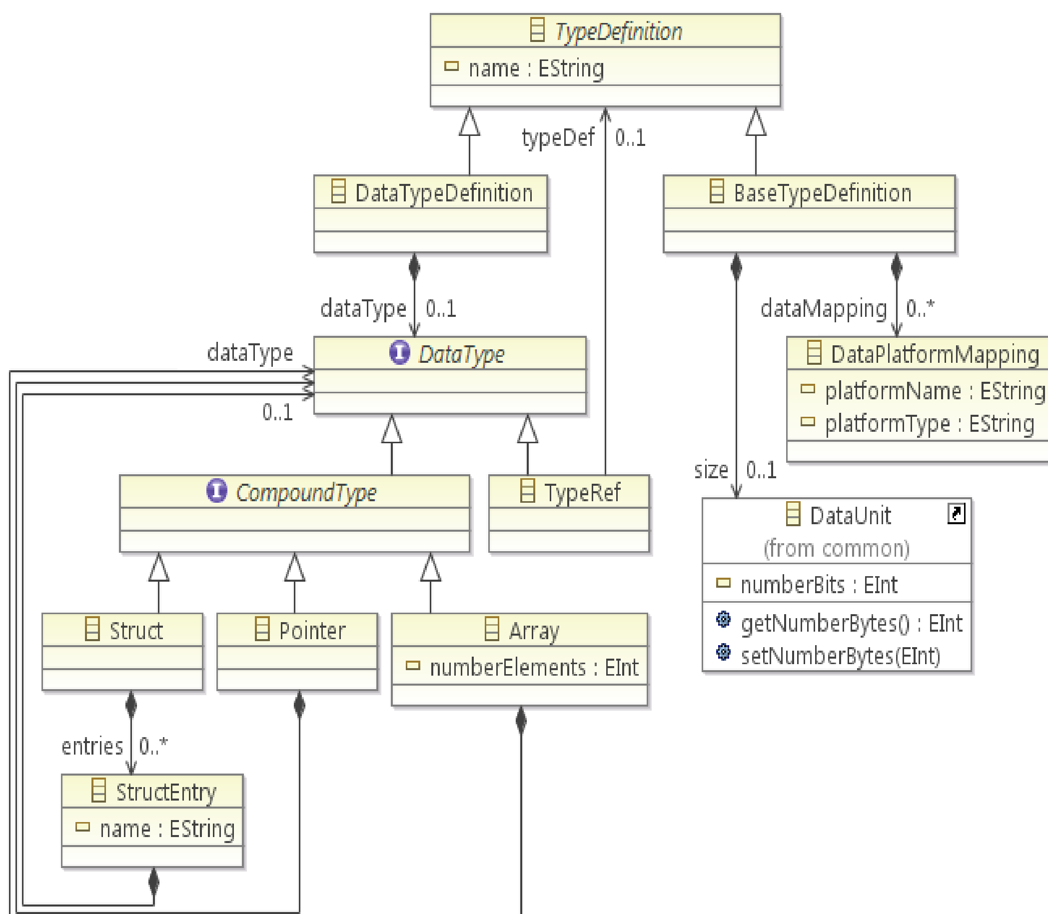

Figure 8.3: Metamodel excerpt for AMALTHEA data types

8.2.1.4 Call graph

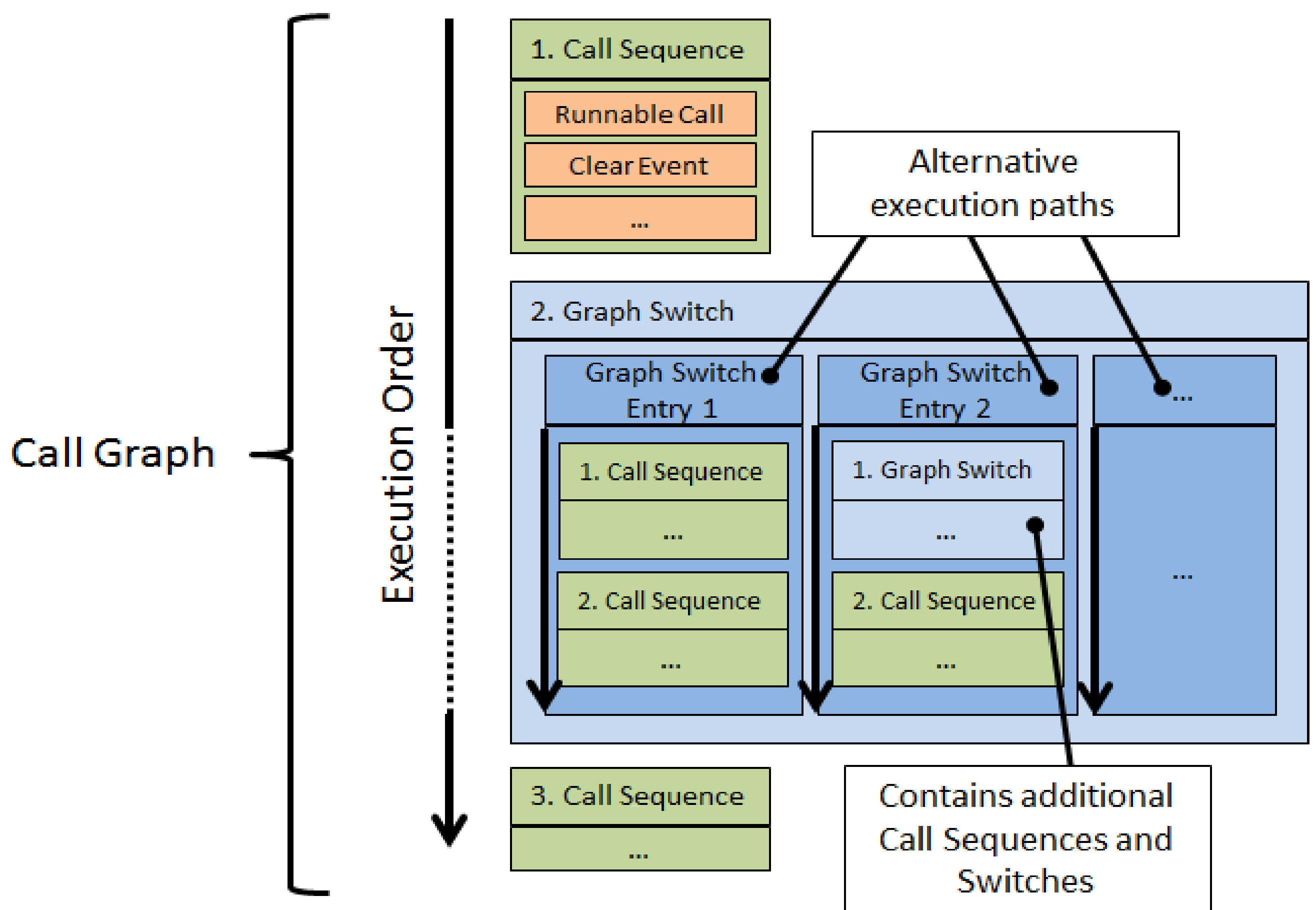

Figure 8.4: Callgraph structure

8.2.2 Components model

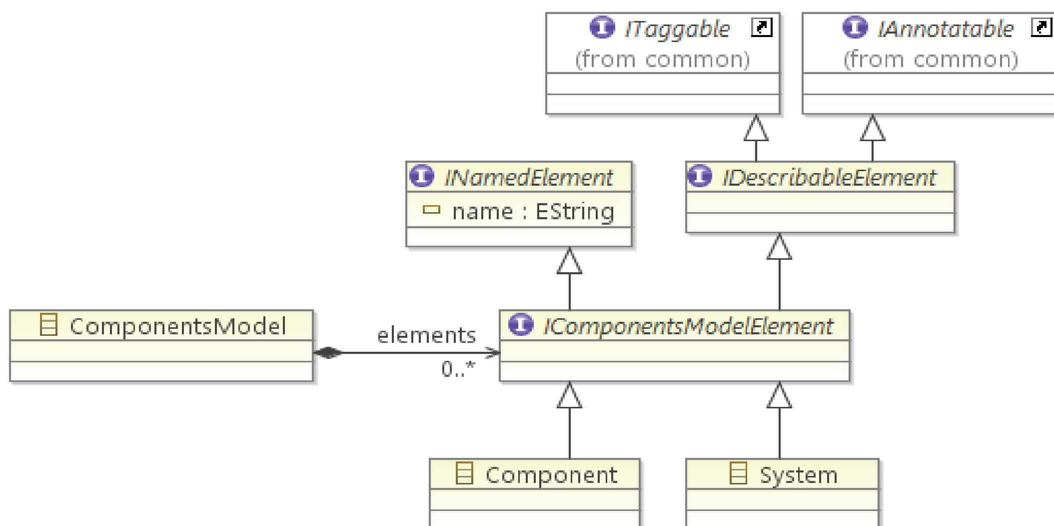

Figure 8.5: Components model in AMALTHEA, which is central accessible through the ComponentsModel type

8.2.3 Runnable items

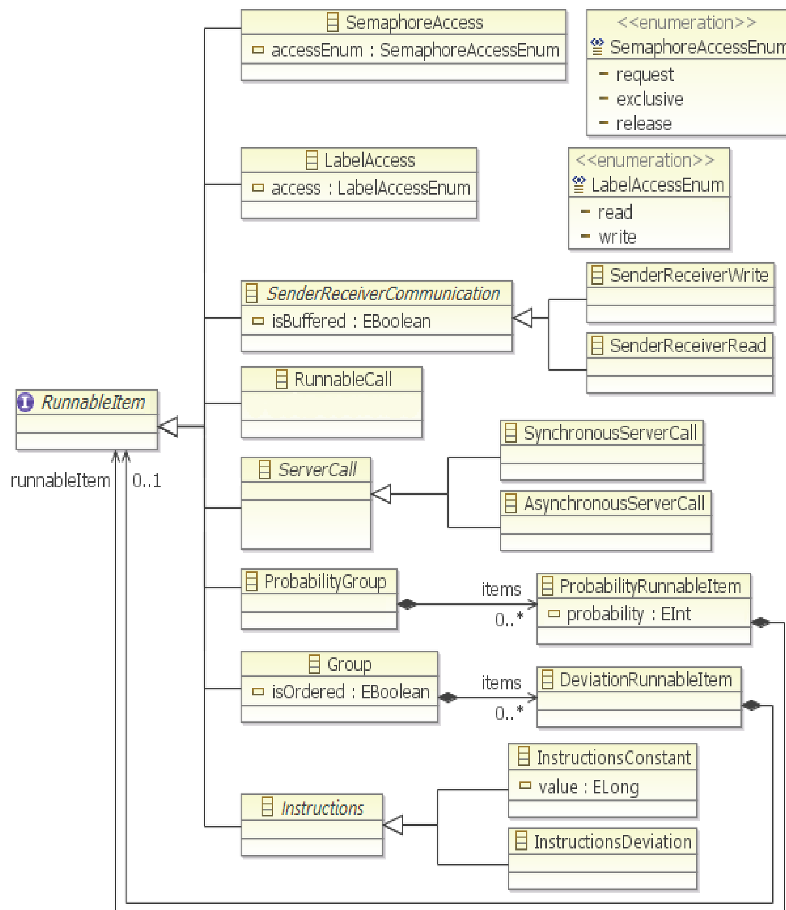

Figure 8.6: The runnable items in AMALTHEA

8.3 ASCET model

ASCET module

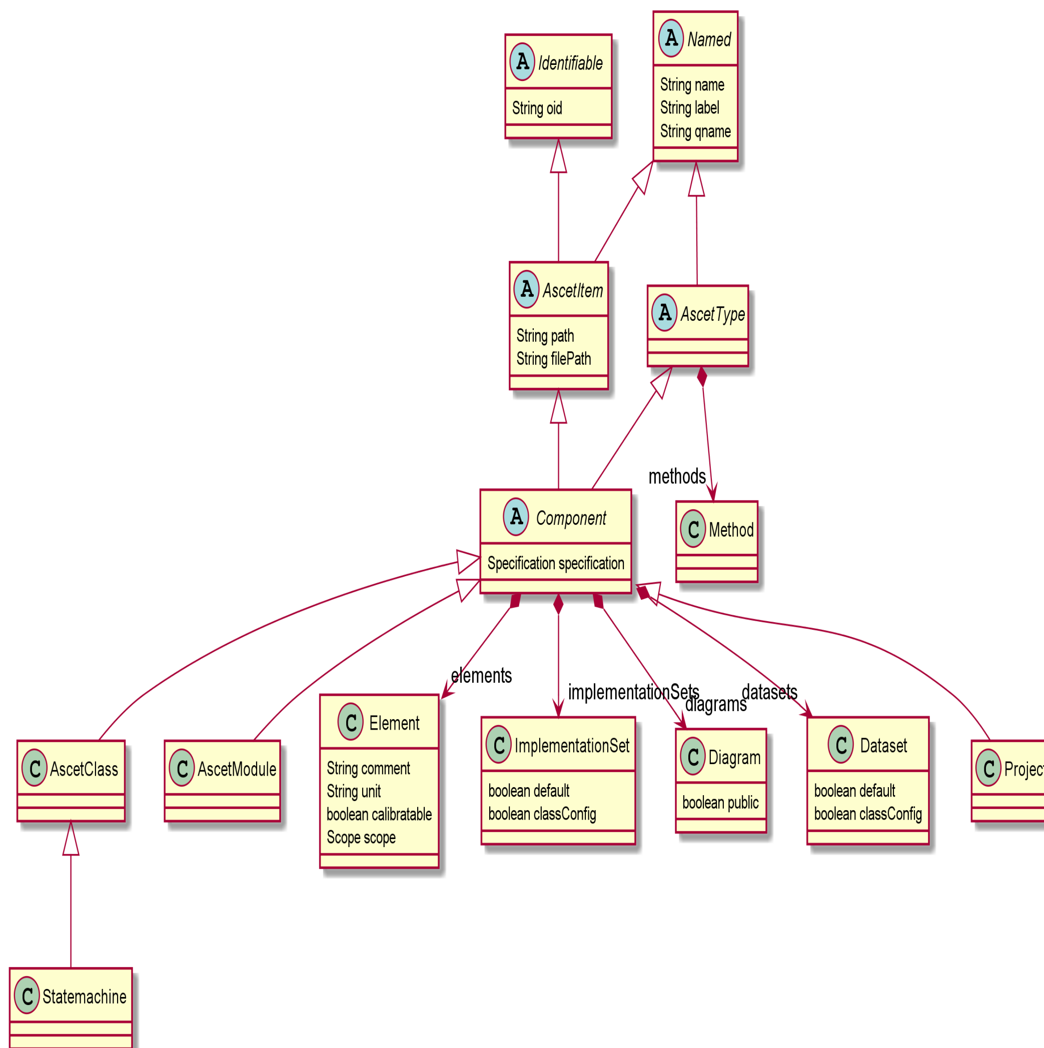

Figure 8.7: The AscetModule type in ASCET

8.4 MIR examples

mapping between SysML Block and AscetModule

```
generates package mir.sysml2ascet //no code is written
generates type SYSML2ASCETs

import package "http://com.bosch.swan.ascet.adom/1.0" as adom
import package "http://www.eclipse.org/papyrus/0.7.0/SysML/Blocks" as sysml

map sysml.blocks.Block as bl and adom.AscetModule as am
{
```

```
when-where {equals(bl. isEncapsulated,true)}
with map sysml.PortAndFlow.FlowPort referencedBy bl.(base_Class).(ownedPort) as
    flowport
       and am.elements(Message) message
{
with map flowport.(name) and message.(name)
when-where equals(flowport.(direction),FlowDirection.In)
with-block {messag.read= true
     messag.written=flase
     }
}
}
//=============================================
map sysml.blocks.Block as bl and adom.AscetModule as am
{
when-where {equals(bl. isEncapsulated,true)}
with map sysml.PortAndFlow.FlowPort referencedBy bl.(base_Class).(ownedPort) as
    flowport
       and am.elements(Message) message
{
      with map flowport.(name) and message.(name)
      when-where equals(flowport.(direction),FlowDirection.Out)
      with-block
        {
        messag.read= false
     messag.written=true
      }
}
}
//================================================
map sysml.blocks.Block as bl and adom.AscetModule as am
{
when-where {equals(bl. isEncapsulated,true)}
with map sysml.PortAndFlow.FlowPort referencedBy bl.(base_Class).(ownedPort) as
    flowport
       and am.elements(Message) message
{
      with map flowport.(name) and message.(name)
      when-where equals(flowport.(direction),FlowDirection.Inout)
      with-block
       {
       messag.read= true
    messag.written=true
    }
}
}
```

Listing 11: Declare the correspondence rule between Task in AMALTHEA and Task in ASCET Using "..." to refer to any order of objects