MODELING VARIABILITY IN COMPONENT ORIENTED SOFTWARE ENGINEERING

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

MUHAMMED ÇAĞRI KAYA

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

FEBRUARY 2015

Approval of the thesis:

MODELING VARIABILITY IN COMPONENT ORIENTED SOFTWARE ENGINEERING

submitted by **MUHAMMED ÇAĞRI KAYA** in partial fulfillment of the requirements for the degree of **Master of Science in Computer Engineering Department, Middle East Technical University** by,

Prof. Dr. Gülbin Dural Ünver _____
Dean, Graduate School of **Natural and Applied Sciences**

Prof. Dr. Adnan Yazıcı _____
Head of Department, **Computer Engineering**

Prof. Dr. Ali H. Doğru _____
Supervisor, **Computer Engineering Department, METU**

Examining Committee Members:

Prof. Dr. Ahmet Coşar
Computer Engineering Department, METU _____

Prof. Dr. Ali H. Doğru
Computer Engineering Department, METU _____

Assoc. Prof. Dr. Halit Oğuztüzün
Computer Engineering Department, METU _____

Assoc. Prof. Dr. Ertan Onur
Computer Engineering Department, METU _____

Dr. Selma Süloğlu
Sosoft Information Technologies _____

Date: _____

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last Name: MUHAMMED ÇAĞRI KAYA

Signature :

ABSTRACT

MODELING VARIABILITY IN COMPONENT ORIENTED SOFTWARE ENGINEERING

Kaya, Muhammed Çağrı
M.S., Department of Computer Engineering
Supervisor : Prof. Dr. Ali H. Doğru

February 2015, 98 pages

A key factor for achieving flexible component oriented applications is to make the components possbile to change and adapt instead of creating and using them from scratch. In order to fulfil various needs in time with little effort, a new metamodel is proposed that establishes a variability specification and system configuration environment for Component Oriented Software Engineering Modeling Language (COSEML). Variability is integrated to COSEML that can be viewed as an Architectural Description Language emphasizing the decomposition view. We refer to this extended version of COSEML as XCOSEML. The textual version of this domain specific language is presented and demonstrated with examples. Moreover, an approach to formally verify component compositions incorporating variability is proposed which eases variability-intensive component oriented system development by reducing complexity of verification. A step by step model transformation approach from XCOSEML to Featured Transition System (FTS) is used which helps generation of FTS models, namely required feature model of the variable component composition and its fPromela specification.

Keywords: Component Oriented Software Engineering, Domain Specific Language, Metamodel, Model Checking, Variability Modeling, Verification

ÖZ

BİLEŞEN YÖNELİMLİ YAZILIM MÜHENDİSLİĞİNDE DEĞİŞKENLİK YÖNETİMİ

Kaya, Muhammed Çağrı
Yüksek Lisans, Bilgisayar Mühendisliği Bölümü
Tez Yöneticisi : Prof. Dr. Ali H. Doğru

Şubat 2015 , 98 sayfa

Bileşen yönelimli uygulamalarda esneklik, bileşenleri değişikliklere olanak verir şekilde tasarlayarak ve sistemler için yeni bileşenler geliştirme yerine var olanları sisteme adapte ederek sağlanabilir. Çeşitli ihtiyaçları zamanında ve az çabayla karşılayabilmek için Bileşen Yönelimli Yazılım Mühendisliği Modelleme Dili (COSEML) için değişkenlik tanımlaması ve sistem konfigürasyon ortamı sunan bir metamodel tasarlanmıştır. Değişkenlik, ayrışma yaklaşımını vurgulayan Mimari Betimleme Dili (ADL) olarak görülebilecek COSEML'ye entegre edilmiştir. COSEML'nin bu genişletilmiş versiyonu XCOSEML olarak adlandırılmıştır. Bu yeni alana özgü dilin metin tabanlı versiyonu örneklerle birlikte anlatılmıştır. Ayrıca, bileşen birleşimlerini değişkenliği de dahil ederek biçimsel olarak inceleyen bir yaklaşım önerilmiştir. Bu yaklaşım doğrulama karmaşıklığını azaltarak değişken yoğunluklu bileşen yönelimli sistem geliştirmeyi kolaylaştırmaktadır. XCOSMEL'den Özellikli Geçiş Sistemleri'ne (FTS) model dönüşümleri adım adım anlatılmıştır. Böylece FTS ile doğrulama yapmak için gerekli değişken bileşen birleşiminin özellik modeline ve davranış dili fPromela'ya dönüşümler gerçekleştirilmiştir.

Anahtar Kelimeler: Alana Özgü Dil, Bileşen Yönelimli Yazılım Mühendisliği, Değişkenlik Modelleme, Doğrulama, Metamodel, Model Kontrolü

To my precious daughter Dîdâr Duru.

ACKNOWLEDGMENTS

Firstly, I would like to thank my advisor Prof. Ali H. Doğru for his constant support, friendship, and guidance. He is a real gentleman and beyond being my advisor, he is a role model both for my academic and family life.

I would like to thank Prof. Ahmet Coşar and Prof. Göktürk Üçoluk who were always interested in my progress. I thank Assoc. Prof. Halit Oğuztüzün and Assoc. Prof. Ertan Onur for their helpful and inspiring comments on my work. I am thankful to Asst. Prof. Onur Pekcan for his encouragement and attention since my education has started at METU.

I thank my dear friend Alperen Eroğlu for all of his technical and moral support during this work. I would like to thank my friend Mahdi Saeedi Nikoo who is one of the kindest and most talented people I have ever met. I also thank my colleagues Mehmet Akif Akkuş, Özcan Dülger, Hüsnü Yıldız, Serdar Çiftçi, Hilal Kılıç, Murat Gençtav, Mine Yoldaş, Gökhan Özsarı, and Alperen Dalkıran. Furthermore, I am thankful to our departmental staff Zafer Şanal, Mehmet Demirdöğen, Sultan Arslan, and Muteber Gökırmak for their friendship and help.

My special thanks go to Dr. Selma Süloğlu who supported me a lot in all phases of my work. When I needed help, she was always there. Sometimes, she postponed her own business to help me. Thank you Selma.

I am grateful to my dear family members; my mother Müzeyyen Kaya, my father Mustafa Kaya, and my sisters Pınar Büyük and Nilüfer Çiçek. Thank you for taking care of me for years and your constant moral support.

Lastly, I would like to thank my dear wife Esra Kaya for her patience during the long working period and her endless support. Thanks Esra.

TABLE OF CONTENTS

LIST OF TABLES

TABLES

LIST OF FIGURES

FIGURES

LIST OF ABBREVIATIONS

ADL	Architectural Description Language
CBD	Component Based Development
CBSE	Component Based Software Engineering
CDL	Component Description Language
CORBA	Common Object Request Broker Architecture
COSE	Component Oriented Software Engineering
COSEML	Component Based Software Engineering Modeling Language
CRG	Component Relational Graph
DSL	Domain Specific Language
EJB	Enterprise Java Beans
FTS	Featured Transition System
SaaS	Software as a Service
SPL	Software Product Line
SPLE	Software Product Line Engineering
SQA	Software Quality Assurance
VP	Variation Point
VPM	Variation Point Model

CHAPTER 1

INTRODUCTION

1.1 Background

Reuse is gaining more importance in software engineering as a development paradigm. Component technology offers reuse rather than building the system from scratch. Components can be independently deployable. Furthermore, they can be integrated with other components to yield a larger system.

Component Based Development (CBD) or Component Based Software Engineering (CBSE) is an approach that suggests reuse of pre-built components in software development. Therefore, software systems can be built faster and the complexity and costs will be reduced. Component Oriented Software Engineering (COSE) approach aims to increase reuse and reduce complexity by applying component-related techniques from a more abstract point of view.

To increase reuse, common and variable parts of a software system should be designated correctly. Variability is the change in software systems or software assets to adapt a different context. Variation Points (VPs) are where the change occurs. There is a set of Variants for each VP. A new system is configured by binding variation points with specific variants.

Software Quality Assurance (SQA) is another important concept for software systems. It is a general concept based on ensuring the quality of software in all development phases. Verification is a sub-concept of quality assurance. We verify systems in order to check for their correct implementations of the user expectations that were

initially defined.

1.2 Problem Statement

The main motivation is to respond to user demands fast and easily in component oriented systems by increasing software reuse and check the final product whether it is consistent and satisfies the user expectations. In component based systems some approaches propose some techniques to solve this issue, however they do not offer sufficient reuse. COSE suggests a purely component oriented approach instead of CBSE. In CBSE generally other techniques of software development (e.g. object-oriented techniques) is used by considering components as development assets. By doing so, a developer cannot use all benefits of component technologies [22]. Instead, all stages of development should be component-aware.

Variability is defined at different levels of development and provides diversity of products by binding different variants to variation points. The more number of available variants increases, the more systems can be built. However, all of these systems are not valid or serve the purpose of user. Therefore, a verification approach should be defined in order to make sure that the system is consistent or provides initial requirements of the user.

1.3 Approach

To fill the gap in using advantages of component systems, we need a variability-centered approach with an explicit representation of variability in COSE. This can be done by specifying variability as a separate model which is then mapped to components, their properties and relations. Our approach enables variability specification in component, composition and interface views. Connector variability is left as a future work. Our hierarchical approach in variability binding is suitable for COSE's top down approach. Moreover, for verification purposes, a model checking technique is applied to variable models in order to check the consistency of models.

1.4 Contribution

The contribution of this thesis is twofold: Firstly, a metamodel for variable component compositions and its realization XCOSEML are introduced. Secondly, a tool is introduced to parse XCOSEML files and aid configuration of XCOSEML models. Also the tool helps the developer to provide a test environment for model checking purposes. Language transformations for model checking are done by the tool.

1.5 Outline of Thesis

Chapter 2 introduces some background information about component technologies, CBSE, COSE and COSE Modeling Language (COSEML). Then, some general information about variability in software systems and variability modeling is given. After stating the importance of variability in component systems, variability in CBSE is given with a review of existing approaches. A comparison of reviewed approaches is also given. Then some background information about verification of software systems is explained based on Featured Transition System (FTS). Finally, the motivation of this thesis is discussed based on the inferences from the literature review.

In Chapter 3 XCOSEML is explained in detail. Smart Home case study is given at the beginning of the chapter. Metamodel and the grammar constructs of the language are given. Then, the tool for XCOSEML is introduced.

Chapter 4 contains the information for verification of XCOSEML models. The reason for choosing FTS for model checking purposes is explained. Then the rules for language transformation are given both for TVL and for fPromela files of the FTS approach. Finally, the verification approach for transformed models is explained.

CHAPTER 2

BACKGROUND

In this chapter some background information is given for component based and oriented techniques. Also, the importance of variability for software systems is discussed and variability in component based systems is explained. At the end of the chapter, the need for variable component oriented systems is stated.

2.1 Component Oriented Software Engineering and COSEML

2.1.1 A brief history for component technology

Component technology is in general, an enabling technology for reuse which is a very important concept in software development. Historically, subroutines were specific parts of the programs whose task was determined in the system requirements. Then, programmers began to reuse these subroutines in implementation of other projects in order to decrease the programming effort [20]. Therefore, the usage of subroutines can be considered as one of the first instances of software reuse concept. Soon after the beginning of wide programming effort, function libraries were introduced as an effective way of reuse.

The following methodology in the history of component technology was structure-oriented development. An application system was divided into modules considering the user expectations. These modules were developed separately, and then they were assembled to yield the application system. When this was accepted, many function libraries were developed as reusable software packages. These packages can be called

as the first generation of software components. However, because of the lack of a methodology, cost-effective software could not been produced [30].

In the 1980s, object oriented technology emerged and reuse became a wider concept. The reuse technology evolved into object oriented class libraries. In 1989, Common Object Request Broker Architecture (CORBA) was introduced as a common reusable middleware [59]. The aim was to develop a middleware for communication of distributed objects of application software without considering location, programming language, operating system, communication protocols and hardware platforms.

The next stage of component technology's history was reusable application framework and platforms. Enterprise Java Beans (EJB) was introduced in 1999 [11]. The motivation behind this technology was to provide application developers with a robust and distributed environment. Meanwhile, COM+ [27] was introduced by Microsoft to support the development of large-scale distributed applications on the windows platform.

Several new methods on component based development have been published [14, 24, 35]. Their aim is to provide engineers with well-defined processes, analysis and specification models, and engineering guidelines.

2.1.2 What is a component?

A component is an implemented software building block that can be composed with other components to yield a software system. Components conform to a standard component model. They can be independently deployed and composed based on a composition standard.

Components are usually pre-written. A component should be independently deployable. In other words, we should be able to compose and deploy a component without having to use other specific components. Each component should have an interface that shows provided services by the component and required services that the component needs to operate correctly. This does not lead to a paradox with the independency rule, because the requires interface does not define how the needed services are provided. Also, components should be well-documented. With the help of this

documentation, potential users can decide whether the component is convenient for their needs [52].

2.1.3 Component Based Software Engineering

The emergence of Component Based Software Engineering (CBSE) has taken place in the late 1990s. The main idea is reusing the software components. CBSE suggests development of software systems from pre-built components rather than building from scratch. Object-oriented development did not lead to extensive reuse, because single object classes were too specific and detailed. Those who wanted to use these classes had to have detailed knowledge about them, even some knowledge about the source code. It was understood that selling or distributing objects as individual reusable components was practically impossible [52].

Components can be classified as in a higher-level abstraction with respect to objects. They have interfaces that describe behavior. Generally, they are larger than objects, but their implementation details are hidden. CBSE is developing software systems by defining, implementing and integrating loosely coupled and independent components. Software systems become larger and complex and the importance of CBSE increases accordingly. To develop dependable software systems fast and deal with the complexity, developers should attach importance to reuse.

CBSE has some fundamental characteristics. For example, components are specified by their interfaces. The interfaces must be clearly separate from the implementation. In this way, implementation of one component can be changed without concerning other parts of the system. Then, components have standards to ease their integration. A component model encompasses these standards. Basically, they define the specifications of component interfaces and how components communicate. With these standards, the operation of components become independent of their programming languages and components written in different languages can be integrated into the same system. As another characteristic, a component based system should have a middleware which provides software support for the integration of components. This middleware copes with low-level issues and lets designers to focus on application-related problems. Also, should have a development process that allows requirements

to evolve, depending on the functionality of available components.

2.1.4 Component Oriented Software Engineering

Component Oriented Software Engineering (COSE) was proposed in [22]. This approach suggests a purely component oriented software development instead of component based approaches. Generally, CBSE approaches are based on object-oriented development but they can also represent components. By using only object-oriented approaches, developers cannot take advantage of all opportunities that component technology offers. All stages of the development are required to be component-aware. The primary concern of the developer should be the composition of the components instead of representation or construction of a component. In component oriented system construction, components are considered as building blocks. Component based approaches are designed to develop systems through programming language statements. The main difference between the two approaches is as follows: component based approaches focus on development, whereas, the focus of component oriented approaches is integration [22].

Modeling activity in COSE starts with a structural decomposition to arrive at existing components. The representation of the system is both in logical and physical levels hierarchically. Figure 2.1 shows a general process model for COSE. The system specification is formed through decomposition and definition of the introduced modules. This results in a connected set of abstract components. Then, components that satisfy the system specifications are located or developed. Finally, these components are integrated and desired software system is composed.

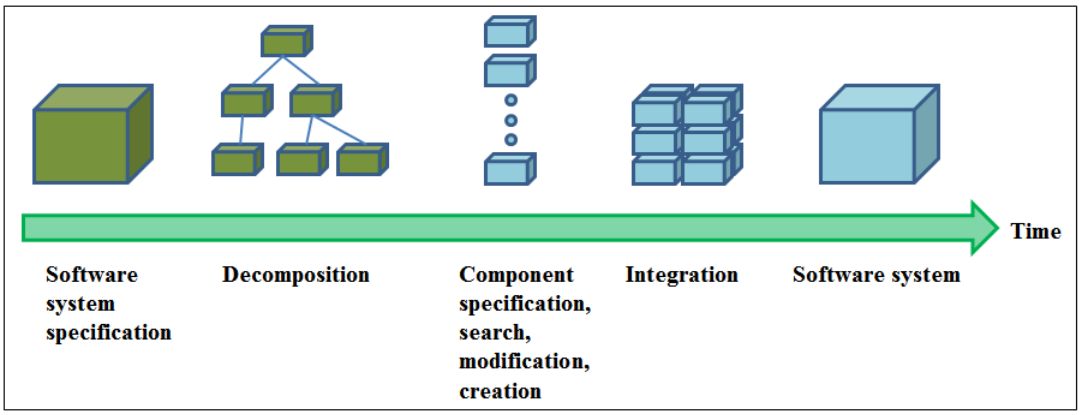

Figure 2.1: A general process model for component based software synthesis [22].

2.1.5 COSEML

Component Oriented Software Engineering Modeling Language (COSEML) [23] is presented to be used within COSE as a graphical modeling language. A modeling language must let the designer to visualize a well suited and graphical decomposition of the system. COSEML has the graphical tools to represent a system as components and their connections. This approach conforms to the idea of developing software by integration rather than code writing. The language provides designers with primitives to represent logical entities and implementation units. Usually, a development process begins with the abstract definitions of system parts. In abstraction level, subsystems are defined first. Then, lower-level components are specified in physical level as the next step. These components implement the responsibilities encapsulated in the previously defined abstract modules. A "represents" link connects an abstract module to its physical implementation. A "connector" symbolizes the connections both among the abstract entities and among the physical entities. Abstract entities of the COSEML are package, data, function, control and connector. The main unit of the physical level is component. Components that only have one interface can also be represented with their interfaces by a special symbol. Moreover, if there are more than one interface that a component provides, then interfaces can be represented in the physical level with their own symbols. Figure 2.2 shows the graphical symbols in COSEML. Therefore, it can be inferred that, a COSEML model can represent a complete model by reconciling both logical and physical components and structural

and operational connections.

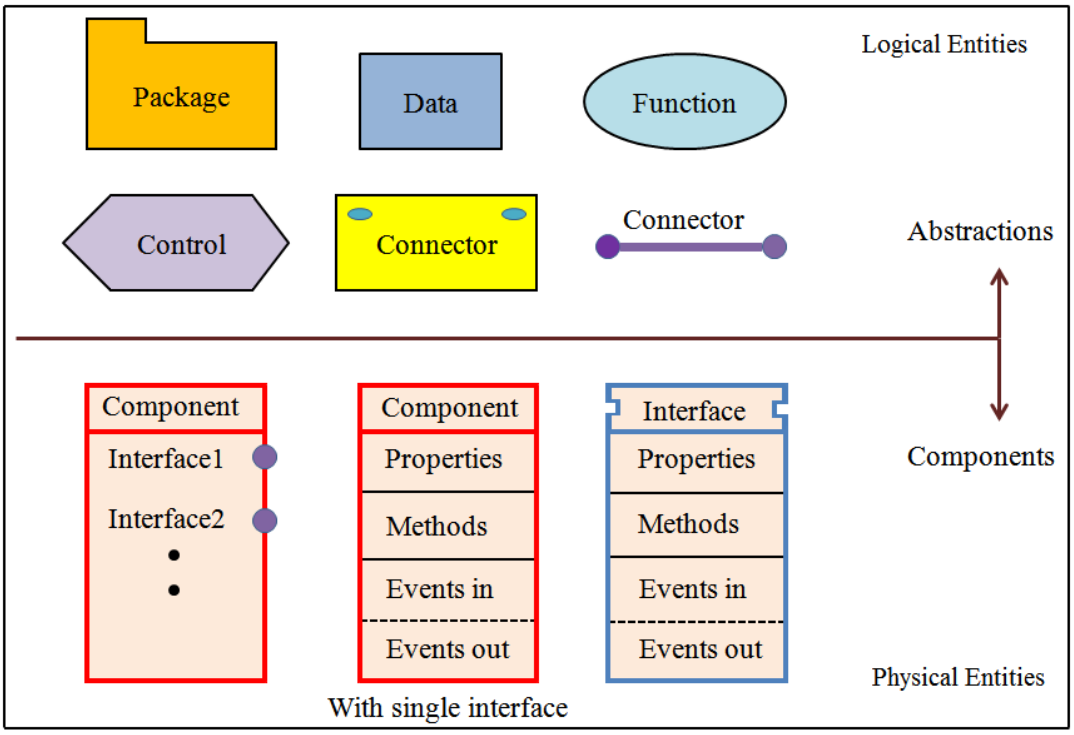

Figure 2.2: Graphical symbols in COSEML [22].

2.2 Variability in Software Systems

2.2.1 Why variability is important for software systems

The notion of variability is gaining a remarkable importance in today's software engineering. In the early days of software technology, development of a system was begun from scratch and systems were relatively static. Any change in a system required editing the source code. Currently, this approach is no longer acceptable. Instead, newer approaches use common parts as much as possible. Design decisions to customize the system are delayed to later stages [58].

Software Product Lines (SPLs), for instance, are based on delayed design decision principle. At the beginning, the final product is not determined. Instead, software architecture is defined and software assets are implemented to match the requirements of a software product family. Run-time adaptive systems are also utilizing the de-

layed design decision concept. They either select the new behavior from embedded alternatives or accept new modules while system is running [44].

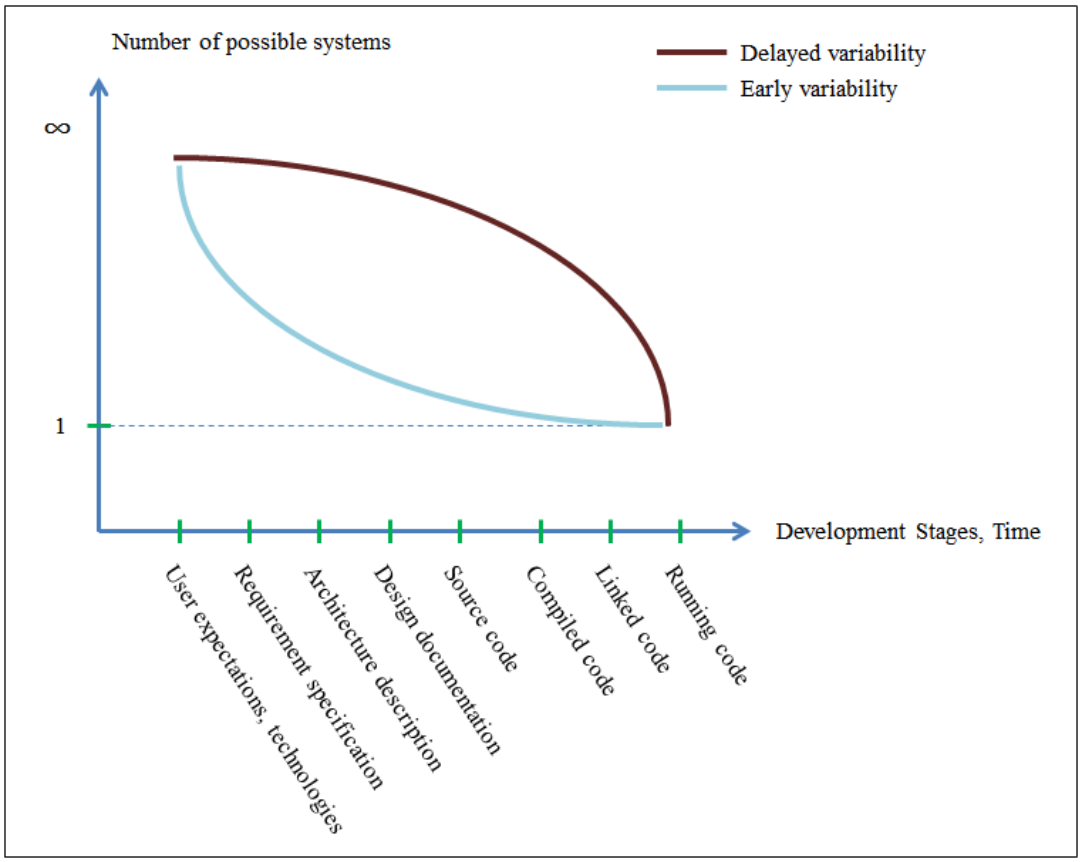

Figure 2.3: Variability change with delayed and early variability (adapted from [58]).

Figure 2.3 describes the constraint of the variability during the development. Initially, system has no constraints, so infinitely many systems can be built. While developers proceed, the number of possible systems is reduced and finally, there is only one system at run-time. At each step, developers make some design decisions and each decision decreases the number of potential systems. In SPLs, delayed design decisions would be beneficial and number of products that uses the same product line assets can be increased. These delayed design decisions can be considered as *variability points*.

2.2.2 Variability Modeling

In software engineering, variability modeling is used to depict variants of a software system in an efficient and formal way. Generally, variability modeling has a close

relationship with SPLs. SPLs manage variability in a systematic way. Self-adaptive systems, open platforms, component-based systems and Software as a Service (SaaS) applications are also designed taking variability into account [51]. When systems consider variability as a principle, reusability of software artifacts and productivity will increase. Although variability brings some advantages, it increases complexity of the system that must be handled with systematic approaches [8, 13, 55]. For all levels of the development, from user expectations to source code, several variability modeling approaches are introduced. Although aims of these approaches are the same, they differ in modeling characteristics, e.g. model choices, abstractions, modeling of quality models, tooling, guidance, and focusing on development activities. In [48], a classification for variability modeling techniques is provided.

Variability is shown in all steps of product family development. Therefore, variability modeling techniques differ in representing the variability in distinct phases. For example, when focusing on the requirement variability, feature models can be used to define commonality and variability in product lines.

In SPL development, variability points are introduced in different levels of abstraction in order to model variability. These levels are architecture description, design documentation, source code, compiled code, linked code and running code. A variability point can be in one of these three states: implicit, designed and bound [58]. When a variability point is shown at a particular abstraction level, this point is also presented at higher levels of abstraction. This is implicit variability. When the design of a variability point is decided in the architectural design phase, it becomes designed. When a variability point binds to a particular variant, it becomes bound. Binding can occur at product architecture, derivation time, compilation time, linking time, start-up time and runtime. Moreover, if new variants can be added to a variability point, it is called *open*. If new variants cannot be added to a variability point, it is called *closed*.

2.3 Variability Management in Component Based Software Engineering

2.3.1 The Need for Variability with Components

Variability in component systems differs from other concepts of variability in different levels. In [40], authors compare component based variability with conventional variability. Although it is not explicitly defined in conventional software development, variability mechanisms are used in terms of variables, parameters, bindings, polymorphism, and configuration. For example, in conventional variability, a variable or function parameter can hold any value of the specified data type. In the case of methods, subclasses can override the method of the superclass. Base method can be substituted with the other version in the subclass, so variability can be defined for methods. For objects the case is similar to the methods. An object can be substituted with an instance of its subclass, so objects have variability.

Variability in component based systems is different from conventional variability. Component based development has a wider range of reusability. In conventional programming, classes and libraries are reused mostly in an application. However, components are developed for inter-organizational reuse. Reuse units of conventional variability are small-grained; however, components are larger reuse units. In conventional variability, variants are assigned at runtime. Nevertheless, in component based development, variants can be set at deployment time that is called "customization". In conventional variability variants can be assigned to a predefined data type (or a compatible data type), so the range of the value of a variant is limited. However, in component based development, completely new and unknown variants can be added, so range of a variant is not limited. Finally, different actors can set the variants in conventional variability. However, in component based development, a developer or a consumer who wants to customize the product sets the variants.

Some approaches use component based techniques integrated to SPLs. In a recent approach Guendouz et al. mention some benefits of component based specification of SPLs [32]. Because both approaches encourage reuse, using them together would be beneficial while developing software. Using component based techniques that provide efficient technologies of development, developers can handle the lack of maturity

in SPL engineering. Moreover, in [57], van der Storm claims that to get benefit from product line approach in terms of reuse, variability must be defined in a component-level concept.

2.3.2 A Review of Existing Approaches

In [43], a new component based variability modeling approach, namely Component Relational Graph (CRG), is proposed. Although the main idea is to manage variability in component models, the configuration logic is not defined explicitly and the approach addresses only component variability.

In [47], authors define a multi-view variability model for business components. To represent variability in functional, static and dynamic view, UML models are extended with stereotypes. Nevertheless, the approach has no mechanism to represent complex nature of variability in composition.

Component Description Language (CDL) is introduced by van der Storm, T. [57] as a new domain specific language for component-level variability. Variability in component and composition is addressed and modeled with CDL.

A hierarchical variability modeling approach is introduced in [33]. They extend MontiArc Architectural Description Language (ADL) with variability related statements. Variability of components are specified locally. Moreover, constraints and variant selections can only be applied on the same or adjacent hierarchical levels. The hierarchical structure of the approach corresponds to component oriented systems. Nonetheless, it has no mechanism to manage variability in composition.

In another approach, Webber et al. introduces a variation point model (VPM) [60]. The approach considers only variability for component specification.

In [10], the proposed approach deals with runtime variability issues dynamically adapting to the changing runtime context. OVM is used as the variability model. Ravazian et al. [46] also use OVM. This approach specifies component, interface, and connector variability using UML. Nonetheless, it doesn't address composition variability.

Kim et al. [40] proposes a variability model in detail which covers five different variability types; variability in attribute, logic, workflow, persistency and interface. Furthermore, three kinds of variability scope are mentioned, namely binary, selection and open scope. Although the approach gives details about workflow variability, the way how to model complex variability relations is not mentioned.

A component-based product line for workflow management systems is introduced in [21]. They use Catalysis method with UML stereotype variability extensions. However, composition variability cannot be modeled with this approach.

Koalish proposes models and configures components and interfaces with explicit variability constructs in [5]. It resolves variability in compile time. However, the approach does not handle composition variability explicitly. Combining Koalish with concepts from feature modeling, Kumbang [4] is introduced as a domain ontology which is developed as a profile extending the UML metamodel.

Luca Gherardi, in his Ph.D. thesis [31], investigates new approaches for development of component based robotic systems. The main concern is to develop reusable and flexible software for robotics. To achive this goal, a software development process is introduced that explicitly takes into account the variability.

Alhalabi et al. [2] propose a UML based approach that lets the system developers to design large-scale systems easily from a high level of abstraction. The integration of components are checked by constraints to ensure consistency of the system. The approach provides a mechanism that is only for component variability.

In [1] authors investigate variability modeling in the video game domain. Their aim is to construct a component based SPL architecture for multiplatform video games using UML to represent product lines. As a drawback, there is no variability mechanism proposed for interface, composition or connector views.

In a recent approach [32], authors combine component based techniques with product line approaches. Their aim is to increase the level of reuse in software development. They define variability for components and connectors.

2.3.3 Comparison of Existing Approaches and Models

Integration of large scale component oriented systems requires more complex handling mechanisms which enable to specify, track, and bind variability and configure the system as expected. Various systems have been introduced in order to address variability management in component oriented systems. Table 2.1 contains a list of existing approaches. Our criteria for the comparison of the approaches is as follows:

- **Variability Modeling:** Defines the model that the approach employs to model variability.
- **Variation Point and Variants (VP&V):** This criterion defines whether the approach has support for variation point and variant specification explicitly. "Yes" indicates that there is an explicit specification; otherwise "No" is stated. If the approach can show variation points as external or internal, then "Ext-Int" is added.
- **Constraints:** If the approach enables to define constraints, then this criterion will be "Yes". Otherwise, "No" will appear in the corresponding cell.
- **Variability In:** Defines where the approach has support variability: Connector (Conn), Component (Cmp), Interface (Int), and Composition (Comp).
- **Tool Support:** Indicates whether there are available tools for the approach.

2.4 Verification of Software Systems

Reliability is critical for software systems, especially for embedded software. *Quality assurance* activity is very important when developing such systems. Quality assurance comprises two aspects: *validation* and *verification*. The aim of validation is to ensure that the developer is building (or built) the system depending on the initial requirements. In other words, validation is to check that if the right system is built. Verification is to make assumptions about the environment and to test whether the system shows certain properties with these assumptions. In other words, to check whether the system is built right [12].

Table 2.1: Comparison of existing approaches.

Approach	Variability Modeling	VP & V	Constraints	Variability In	Tool Support
de Souza Gimenes et al. [21]	UML extension	Yes	No	Cmp	No
Webber et al. [60]	UML extension	Yes	Yes	Cmp, Comp, Int	No
Kumbang [4]	Feature Model	No	Yes	Cmp, Int	Yes
Kim et al. [40]	new VM	Yes	No	Cmp, Comp, Int	No
Saidi et al. [47]	UML extension	Yes	Yes	Cmp	No
Razavian et al. [46]	OVM	Yes, Ext/Int	Yes	Cmp, Conn, Int	No
Bencomo et al. [10]	OVM	Yes, Ext/Int	Yes	Cmp, Int	No
Van der Storm [57]	CDL	No	Yes	Comp	Yes
Haber et al. [33]	new VM	Yes	Yes	Cmp	No
İleri et al. [43]	new VM	Yes	Yes	Cmp	No
Koalish [5]	new VM	No	Yes	Cmp, Int	Yes
Luca Gherardi [31]	Feature Model	No	Yes	Cmp, Comp	Yes
Alhalabi et al. [2]	UML extension	No	Yes	Cmp	No
Albassam et al. [1]	UML extension	Yes	Yes	Cmp	No
Guendouz et al. [32]	new VM	Yes	No	Cmp, Conn	No

We used *model checking*, that is one of the verification techniques, to test our models. Model checking is mainly concerned with *models* and *properties*. Models describe the behavior of a system. Properties are the system specifications that are checked if the system satisfies them or not. The task of model checker is, when it is given a model and a property, to determine whether or not the model satisfies the property.

The behavior of a system must be described for model checking. In the basic form, a system's behavior can be described with *states*. A state, for a system that contains a single program, can be exemplified as a snapshot of the memory block allocated to the program. All states of the system compose the *state space*. Model checking algorithms make searches in a system's state space. Therefore, the model checking problem can be described as a decision problem that consists of determining whether a transition system satisfies a property [15].

Variability lets the developers to customize their product and offer it to a wider market. In the case of Software Product Line Engineering (SPLE), many software systems or products are produced to benefit from common parts of these systems as much as possible. The differences among the products are called the *variability*. This kind of systems in which variability has a great importance is *variability-intensive systems*. Building variability-intensive systems has many advantages, such as productivity gains, shorter times to market and wider market coverage [19, 45].

In SPLE differences among products are expressed in terms of features. Model checking of SPLs gets harder when the number features increases. *Featured Transition Systems* (FTSs) are extensions of transition systems. They are used as a formalism to describe the behavior of all the products of an SPL. In FTSs, transitions are labeled with features.

In [15], the algorithms for FTS model checking are introduced, such as semi-symbolic fLTL model checking algorithms and fully symbolic fixed-point based algorithms for fCTL model checking. SNIP is introduced by Classen et al. as a tool for model checking for SPLs [17]. Classical tools are only capable of checking properties for each product one by one. However, SNIP can check all products of the SPL in a single step with its specifically designed algorithm. An SPL is specified on SNIP with the combination of two specification languages; *TVL* and *fPromela*. TVL is used to

describe the variability in the SPL and fPromela is used to describe the behavior of individual products. The authors claim that SNIP is one of the first tools equipped with specification languages to formally express the variability and the behaviors of the product of SPL together.

2.4.1 Verification of Component Based Systems

Reuse of components in software systems is increasing. However, it is possible to confront some side-effects because of unique characteristics of the components. Quality assurance and testing is crucial for this cases. In [30], authors investigate potential strengths and weaknesses in component based methodologies and they focus on testing for quality assurance purposes.

In a software system, components work in a black box nature and component users cannot reach the source code. This causes some problem when testing the whole system. Component developers test the components themselves, independent of the application of the user. So, testing the component alone is not a solution to this problem. In [34, 50, 61], authors refer to this problem and propose some techniques as solutions.

In [62], a hybrid verification approach is proposed for verification of component based systems that covers both model checking and traditional software testing. In another approach [38], a tool supported technique is used for verification of component based systems by model checking.

Above-mentioned approaches for verification or testing component systems do not work on variability-intensive models. In [37], an approach is proposed for product line model checking to support maintenance after the evolution of a product line. In [56], the mCRL2 language is used for behavioural variability analysis of component based product lines.

2.5 Discussion

In section 2.1.4 we explain the differences between COSE and CBSE and we claim that we need COSE techniques to efficiently use all benefits of component technologies. To this end, we develop a component oriented variability-intensive architectural language. Although there are a number of researches about component based technology, there is a significant leap in component oriented techniques.

Our modeling approach is different from many researches in the literature in terms of a separate variability model. We then map the variability with the components and other related assets. Moreover, our systematic approach defines the variability for package, component, composition, and interface. Again, many approaches lack an explicit variability definition for all these assets. However, in our approach, the connector variability is missing. Even though our current metamodel provides an infrastructure to support this kind of variability, no mechanism is provided for now.

We review some testing and verification approaches in section 2.4.1. However many of these approaches do not take variability into account. Moreover, many component based variability modeling approaches do not have verification tests. In a recent systematic literature survey [29], existing research on variability in software systems including component based systems is analyzed. In the assessment part of the work, one of the results of the study indicates that existing approaches lack validation mechanisms. Therefore, verification of variable component systems are important.

CHAPTER 3

VARIABILITY IN COSE: XCOSEML

XCOSEML is an extension of Component Oriented Software Engineering Modeling Language (COSEML). XCOSEML aims to incorporate variability with component oriented development in a hierarchical way [39, 54].

In this chapter, a case study is introduced in XCOSEML. Then, a metamodel for XCOSEML is introduced and XCOSEML language constructs are specified in detail. Finally, tool support for XCOSEML is explained.

3.1 Case Study: Smart Home

Smart home system is used as a case study to illustrate the application of our verification approach. Smart home system is a set of utilities provided to home residents for their time, energy, and money savings and comfort that can be controlled remotely by a smart phone or a program. These utilities communicate with the automated home devices and sensors such as for lighting, heating, air conditioning, audio and video consoles, cameras, and wearables. The system incorporates diverse home devices, enables cross communication and provides a monitoring and management environment. For instance, the system can automatically steep coffee, prepare favorite play lists, generate energy status report, check and report access history during the day and unlock the doors when the home resident is close enough to the entrance.

With a variety of features, smart home system can include a set of sub-domains each of which covers domain related functionalities, namely security, safety, telecare,

entertainment, energy management and hobby garden. Security deals with camera surveillance, locking and unlocking doors and garage (if exists), notification mechanism in case of emergency and authentication and authorization related functionalities. Safety enables a management structure where home devices and lightning can be turned on and off, devices are monitored, anomaly is detected, control and early notification of hazards are achievable. Telecare covers monitoring and alerting facilities of home resident's health status by an array of wearables (e.g. smart watches, wrist bands, and smart clothes), fall detection, medicine reminding and preparation of exercise program with regard to her/his health status. Entertainment controls audio and media devices, tracks home residents' favorite play lists, establishes tele-conference sessions with medical team (e.g. doctor, physician) for consultation or control purposes. All facilities related with controlling, monitoring and reporting energy resources (e.g. water, wind panels, electricity) are in the scope of energy management. Hobby garden includes monitoring of needs and changes in soil and plants with respect to temperature and humidity. These sub-domains and their functionalities vary according to home resident's needs and the size of the home (small, big and large). For instance hobby garden related features can only exist if the home has a garden itself. What's more it is not a must that the smart home system includes all aforementioned subdomains in one place. Smart home system covering a part of functionalities with a set of variability is represented in XCOSEML. Figure 3.1 shows the decomposition of the Smart Home system in COSEML.

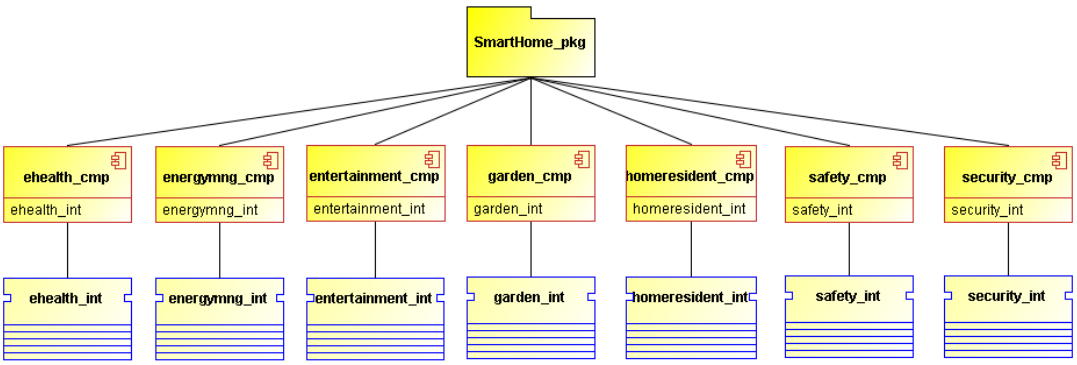

Figure 3.1: Smart Home system in COSEML.

3.2 Metamodel of XCOSEML

The metamodel is introduced to help modeling and integration of variability-intensive component oriented systems. Our metamodel integrates COSEML and variability specifications. Fundamentally, the metamodel enables to define system decomposition in a top down manner along with package and component variability. An overview of the metamodel is presented in Figure 3.2. It basically shows package and component main blocks.

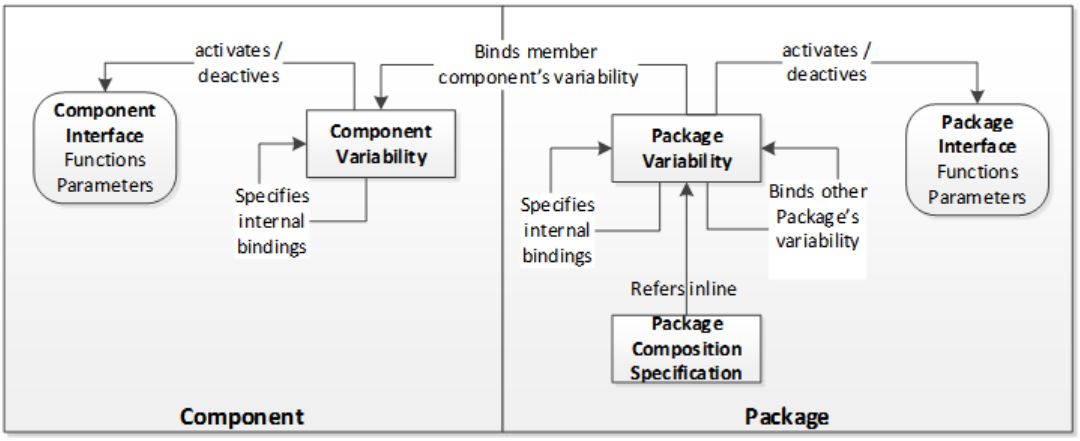

Figure 3.2: Overview of the metamodel.

In our hierarchical variability approach, the configuration rules of below levels can be determined and managed only by preceding upper levels. A package deals with variability of its member packages or components. At the same time, variability of a package or component is managed by its upper level. Reducing complexity by describing variability in a decentralized way is the main benefit of this approach.

Variability in a package involves and binds the variability of its member packages and components. Moreover, a package's variability can manage the configuration of its member component interface by activating or deactivating of methods and setting or unsetting of parameters. Thus, different packages can have different component interfaces of the same component. This provides reusability for components and interfaces. Moreover, components can change their interfaces based on their own variability bindings.

In composition of a package, the way of how sub-packages or member components interact with each other is introduced. Inline references of its variability are included in order to determine the changeable parts of the composition. These references provide a way to model a set of possible required behavior to satisfy different composition needs.

The metamodel is described in Figure 3.3 which contains three main parts. The left-most part depicts both dynamic and static views of COSEML. In static view, building blocks of component oriented systems, i.e. packages (for logical level) and components and their interfaces (for physical level), are shown. Dynamic view covers composition of packages and components that contains a set of messages and interactions. Variability specifications and their mappings to COSEML (the mid and right-most parts) are adapted from [54]. Packages and components can define their variation points both in external and internal views. Also, numerical or logical constraints can be defined for these variation points. Configuration variation points are abstractions for internal and external variation points. They hide the details of low level variability bindings.

Our metamodel provides variability for packages, components, interfaces and compositions. Connector is modeled as a set of messages that defines a connection among methods in COSEML specification. Variability of connectors corresponds to message variability in our model. Although, our metamodel lets the definition of message variability, no mechanism is provided for now and it is postponed as future work.

3.3 XCOSEML Language

XCOSEML is a new domain specific language that is an extension of COSEML. It has been developed based on the metamodel given in Figure 3.3. The language is developed using *Xtext* [9, 25, 28]. Xtext is integrated to Eclipse IDE and provides a development environment for domain specific languages. There are five different XCOSEML models: *package*, *component*, *component interface*, *configuration interface*, and *composition specification*. The full content of XCOSEML grammar can be found in Appendix A.

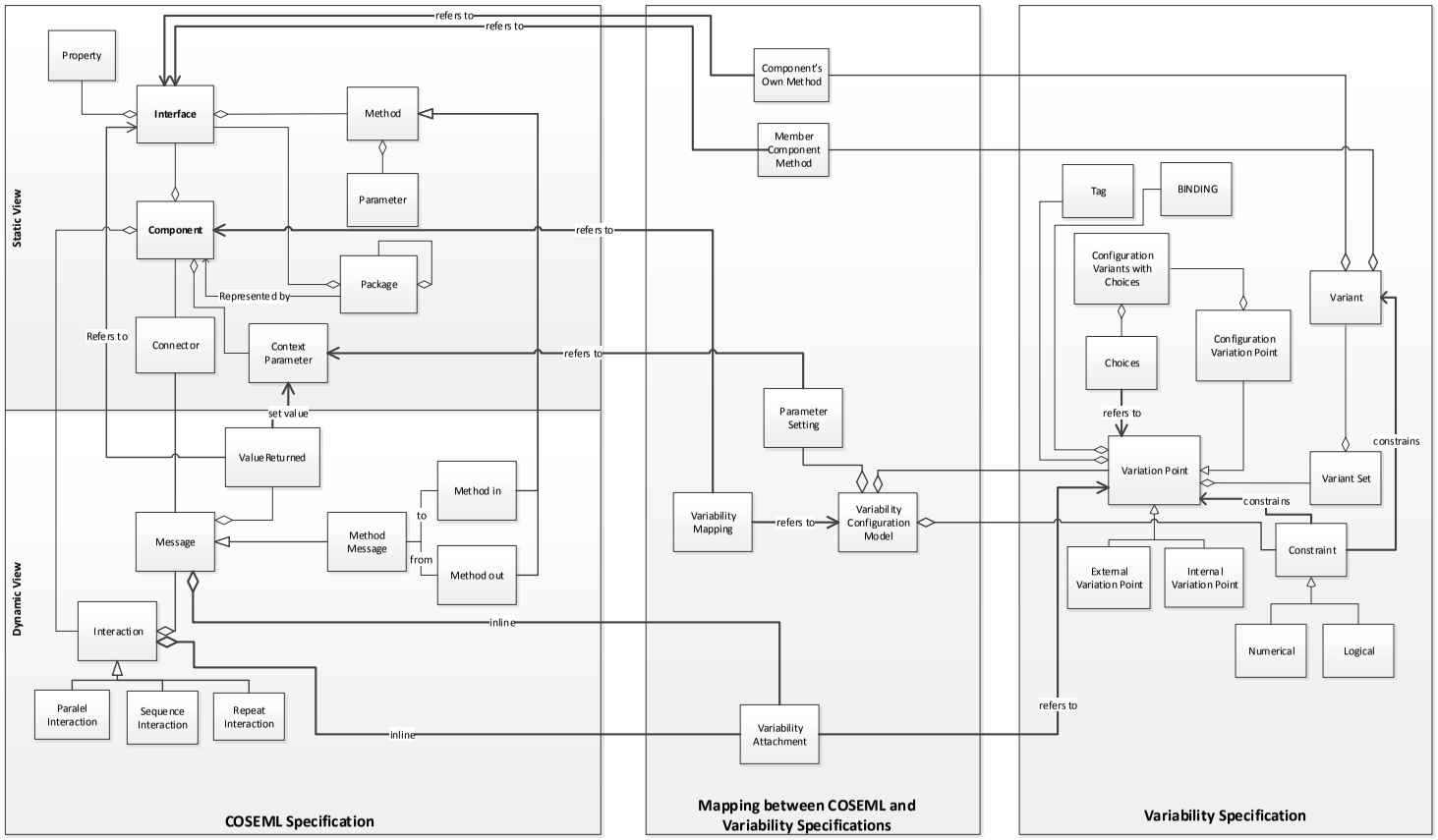

Figure 3.3: XCOSEML metamodel.

3.3.1 Variation Specification Constructs

Variation specification constructs are elaborated with Xtext specification. XCOSEML metamodel of variation point specification constructs are given in Figure 3.4.

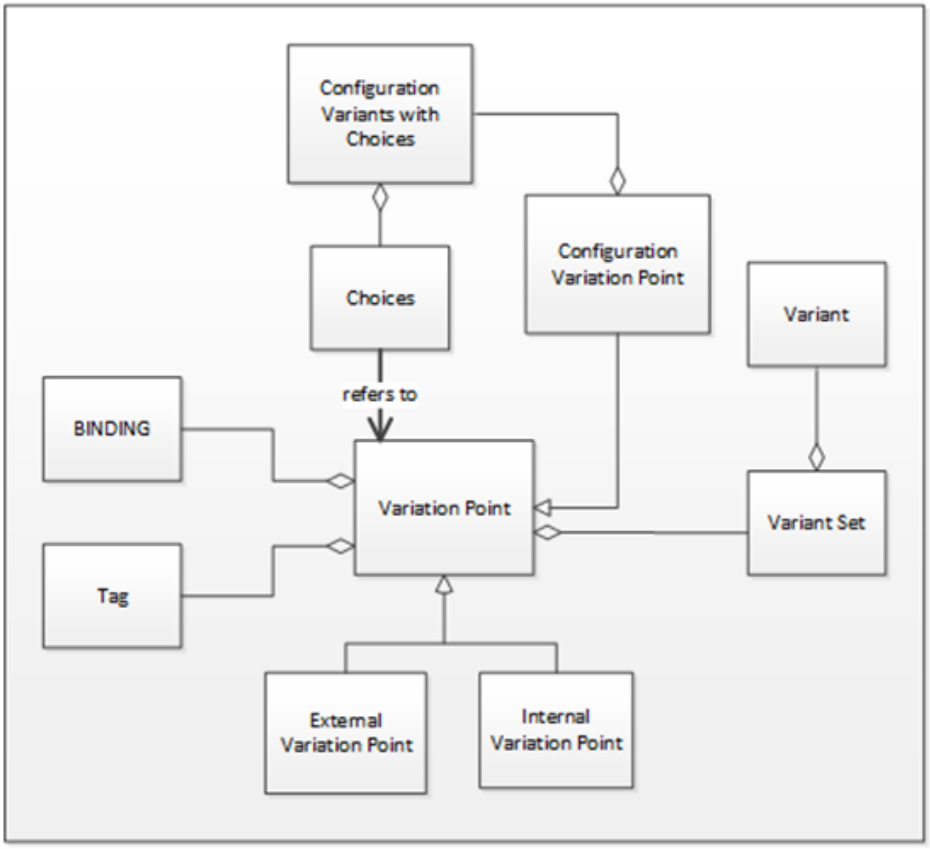

Figure 3.4: Variation Point Specification Constructs of XCOSEML metamodel.

VarPoint: VarPoint refers to variation point that can be described as where differences among products occur. VarPoint is a high level abstraction of *internal*, *external*, and *configuration* variation points. Xtext specification of VarPoint is given in Figure 3.5.

```
VarPoint:
      ConfigurationVarPoint | InternalVarPoint | ExternalVarPoint
;
```

Figure 3.5: XCOSEML VarPoint Representation in Xtext.

InternalVarPoint: This type of variation points is invisible to outer context. It describes a variability with a set of variants (VariantSet) and specified binding time (BINDING). They can be configured in configuration interfaces of packages or components. Figure 3.6 shows InternalVarPoint in Xtext.

```
InternalVarPoint:
      vt = "internalVP" name=ID ':'
      variants = VariantSet
      "bindingTime" btime =BINDING
;
```

Figure 3.6: XCOSEML InternalVarPoint Representation in Xtext.

ExternalVarPoint: These variation points are visible to outer context. They can be configured by other packages or components. They specify their variability with a set of variants (VariantSet) and specified binding time (BINDING). Figure 3.7 shows ExternalVarPoint in Xtext.

```
ExternalVarPoint:
      ( vt = "externalVP" | vt2 = "vp")  name=ID ':'
      variants = VariantSet
      "bindingTime" btime =BINDING
;
```

Figure 3.7: XCOSEML ExternalVarPoint Representation in Xtext.

VariantSet: Three group of variants can be defined that are *mandatory*, *optional*, and *alternative*. For alternative variants, we can specify the number of variant selections by defining minimum and maximum limits. Xtext grammar description of VariantSet is in Figure 3.8.

```
VariantSet:
      {VariantSet} ("mandatory" (variants += Variant)* )?
      ("optional " (variants += Variant)*)?
      ("alternative " (variants += Variant)*
          "(min:" INT ",max:"INT ")")?
;
```

Figure 3.8: XCOSEML VariantSet Representation in Xtext.

Variant: A variant is a determined instance of a variable property. Variants can activate methods of functions and set parameters that are specified in component interfaces. Figure 3.9 refers to the grammar description of Variant.

```
Variant returns Variant :
      "variant" name = ID  (
            (":activateMethods("
                  funct = [MethodIn] ("," functs += [MethodIn])* ")"
            )?
            (":setParameter(toFunct:" f = [MethodIn]
                  ",parameter:" pars = Param
                  (";toFunct:" func += [MethodIn]
                   ",parameter:" fpars += Param)*")"
            )?
      )
;
```

Figure 3.9: XCOSEML Variant Representation in Xtext.

Tag: A tag indicates where a variation point resides. It can reside in a composition, a configuration interface, or a configuration variation point realization. Figure 3.10 shows the Xtext grammar of Tag.

```
Tag:
      "@" (name = "composition" |
           name = "vconfiguration" |
           name = "vconfrealization")
;
```

Figure 3.10: XCOSEML Tag Representation in Xtext.

Binding: Binding indicates when a variation point binds to a variant. Figure 3.11 shows the options in Xtext environment.

```
BINDING:
      devt = "devtime" | derv = "derivation" | comp = "compilation" |
      link = "linking" |strt= "start-up" | runt ="runtime"
;
```

Figure 3.11: XCOSEML Binding Representation in Xtext.

In table 3.1 examples of internal and external variation points are given. Examples

Table 3.1: Examples for Internal and External Variation Points.

```
internalVP bedroom:
 optional
  variant onebedroom
  variant twobedroom
  variant threebedroom
  variant morethanthree
bindingTime devtime
```

```
externalVP functionalities:
 mandatory
  variant security
  variant safety
 optional
  variant telecare
  variant hobbygarden
  variant entertainment
  variant energymanagement
bindingTime devtime
```

are taken from *security_conf* configuration interface of Smart Home case study.

ConfigurationVarPoint: An abstract high level variation point definition that maps its variants to a set of internal variation points with their variant selections, specifying each realization. It can be either internal or external which is specified by "varType" keyword. It defines a set of variants (VariantSet) and their realization (ConfVariantWithChoices), default variant (Variant) selection and binding time (BINDING). A ConfigurationVarPoint structure in XCOSEML grammar is shown in Figure 3.12.

```
ConfigurationVarPoint returns ConfigurationVarPoint:
      "configuration" (
            {InternalVarPoint} name=QualifiedName ':'
                  "varType"  vt = "internalVP" |
            {ExternalVarPoint} name=QualifiedName ':'
                  "varType"  vt = "externalVP"
      )
      (variants = VariantSet)
      ("realization" rea = STRING)
      ((confvariants += ConfVariantWithChoices)+ )
      ("defaultVariant" defaultVariant = [Variant])
      ("type" type= CONFTYPE "bindingTime" btime = BINDING )
;
```

Figure 3.12: XCOSEML ConfigurationVarPoint Representation in Xtext.

ConfVarWithChoices: A variant definition of a configuration variation point including a set of choices. The grammar definition is given in Figure 3.13.

```
ConfVariantWithChoices:
       "confvariant" name = ID "mapping"
       (choices += Choice)+
;
```

Figure 3.13: XCOSEML ConfVarWithChoices Representation in Xtext.

Choice: *Choice*, whose definition is given in Figure 3.14, is the selection definition of a variation point with its selected variants. Optionally, minimum and/or maximum number of variants can be defined.

```
Choice:
       "VPName" vp = [VarPoint]
       "selectedVariants("
          (vars += [Variant])+  ("; min:" INT)? (", max:" INT)?
       ")"
;
```

Figure 3.14: XCOSEML Choice Representation in Xtext.

Table 3.2 shows a configuration variation point taken from *smarthome_conf* configuration interface of Smart Home case study. *homesize* variation point presents a high level configuration structure. It has two alternative variants; *small* and *middlerange*.*small* variant is realized by variants *onebedroom* or *twobedroom* of *bedroom* variation point (line 10). Only one parameter can be selected because of the given constraint (min:1,max:1). The default variant of this configuration variation point is *small* (line 12) and it binds to a variant at development time (line 14).

Constraint: With the grammar definition in Figure 3.16, *Constraint* is an abstraction of logical (LogicalConstraint) and numerical (NumericalConstraint) constraints. Figure 3.15 shows the constraint specification constructs of XCOSEML metamodel.

Table 3.2: A Configuration Variation Point from Smart Home Case Study.

```
1  configuration homesize:
2   varType externalVP
3   alternative
4    variant small
5    variant middlerange
6    variant big
7    (min:1,max:2)
8   realization "determined by bedroom size"
9   confvariant small mapping
10   VPname bedroom selectedVariants( onebedroom
   twobedroom; min:1,max:1 )
11  ...
12  defaultvariant small
13  ...
14 bindingTime devtime
```

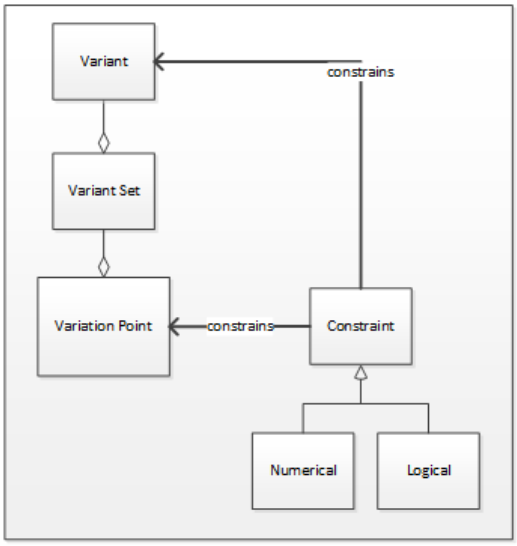

Figure 3.15: Constraint Specification Constructs of XCOSEML Metamodel.

```
Constraint:
      LogicalConstraint | NumericalConstraint
;
```

Figure 3.16: XCOSEML Constraint Representation in Xtext.

LogicalConstraint: *LogicalConstraint* is a definition depicting a constraining relationship in which a variation point and/or related variants decide another variation points and/or its selected variants status either *excluded*, *implied*, *required* or *negated*. XCOSEML representation of LogicalConstraint is given in Figure 3.17.

```
LogicalConstraint:
      ( p1 = [VarPoint] (p2 = [Variant])? )
      c =CONST  p3 = [VarPoint]
      ("selectedVariants("(vars += [Variant])+
            (", min:" INT)? (", max:" INT)? ")"
      )?
;
```

Figure 3.17: XCOSEML LogicalConstraint Representation in Xtext.

NumericalConstraint: *NumericalConstraint* is a definition depicting a constraining relationship in which a variation point and related variant result in an assignment of a value to another variation point and related variant or to a property with expressions greater than (>), less than (<), greater than or equal (>=), less than or equal (<=), equal (==), not equal (!=). The grammar definition is shown in Figure 3.18.

```
NumericalConstraint:
      lhs=RHS "const" rhs = RHS exp = EXPR
      (STRING |  "valueOf{" (vars += [Variant])* "}")
;
```

Figure 3.18: XCOSEML NumericalConstraint Representation in Xtext.

3.3.2 Static View: Package, Component, Interface Constructs

This part contains definitions of static constructs and some related constructs of XCOSEML language. Figure 3.19 contains corresponding metamodel.

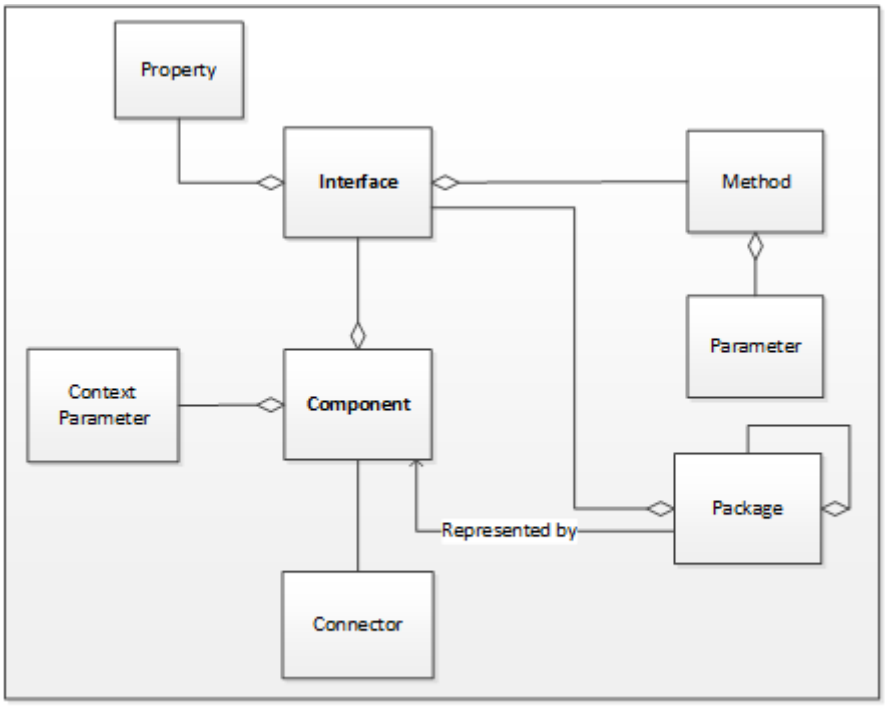

Figure 3.19: Metamodel of XCOSEML Static Constructs.

Package: A *Package* includes its member *Constructs* (one and more) and contains its own *Interface*. A *Construct* refers to either a *Package* or a *Component*. Moreover, a package can have configuration interface (*VarConfigurationModel*) and compositon specification (*CompositionSpecification*). Figure 3.20 shows the XCOSEML representation of package in Xtext environment.

```
Package:
       "Package" name = QualifiedName
       ("includes" (element += [Construct])+ )
       ("Interface" (int = [Interface]))
       ("Configuration Interface" (confint = [VarConfigurationModel]))?
       ("Composition" compname = [CompositionSpecification])?
;
```

Figure 3.20: XCOSEML Package Representation in Xtext.

QualifiedName and Construct: *QualifiedName* specifies that how names of elements should be defined. A *Construct* refers to either a *Component* or a *Package*. Grammer definitions are given in Figure 3.21.

```
QualifiedName:
	ID ('_' ID)*

;

Construct:
	Component | Package
;
```

Figure 3.21: XCOSEML QualifiedName and Construct Representations in Xtext.

Component: Definition of a *Component* contains the name of the component, component's interface and configuration interface of the component if there exists. Figure 3.22 shows the grammar definition.

```
Component:
	"Component" name = QualifiedName
	("Interface" (int = [Interface]))
	("Configuration Interface" (confint = [VarConfigurationModel]))?
;
```

Figure 3.22: XCOSEML Component Representation in Xtext.

Interface: *Interface* is an abstraction of a "component interface" or a "package interface". The grammar definition given in Figure 3.23 indicates that the name of the interface must be defined. Designer can define any number of property for an interface (*CompProperty*). CompProperty is a variable definition that belongs to a component. It is composed of simply a name (*ID*). An interface must provide at least one method (*MethodIn*). Also, an interface may require some methods (*MethodOut*).

```
Interface:
	"Interface" name = ID
	("Properties" (compproperties += CompProperty)*)?
	("Provided Methods" (methodsin += MethodIn)+)
	("Required Methods" (methodsout += MethodOut)*)?
;
```

Figure 3.23: XCOSEML Interface Representation in Xtext.

Method: A method is an abstraction for *MethodIn* or *MethodOut*. The grammar definition is in Figure 3.24.

```
Method:
	MethodIn | MethodOut
;
```

Figure 3.24: XCOSEML Method Representation in Xtext.

ContextParameter: A *ContextParameter* refers to a shared element used in component composition. The related grammar definition is in Figure 3.25.

```
ContextParameter:
	name = QualifiedName (defaultvalue = INT | STRING | ID | BOOLEAN)
;
```

Figure 3.25: XCOSEML ContextParameter Representation in Xtext.

3.3.3 Dynamic View: Composition Specification Constructs

CompositionSpecification: *CompositionSpecification* includes its configuration interface (VConfModelImport) if it exists. Then, imports its components (ComponentImport) and defines shared variables (ContextParameter). It maps component variability onto related variation points and variants (VMMapping). Also, it defines the compositon for each method. Composition specification part of the metamodel is depicted in Figure 3.26. Figure 3.27 shows the grammar definiton for a composition specification.

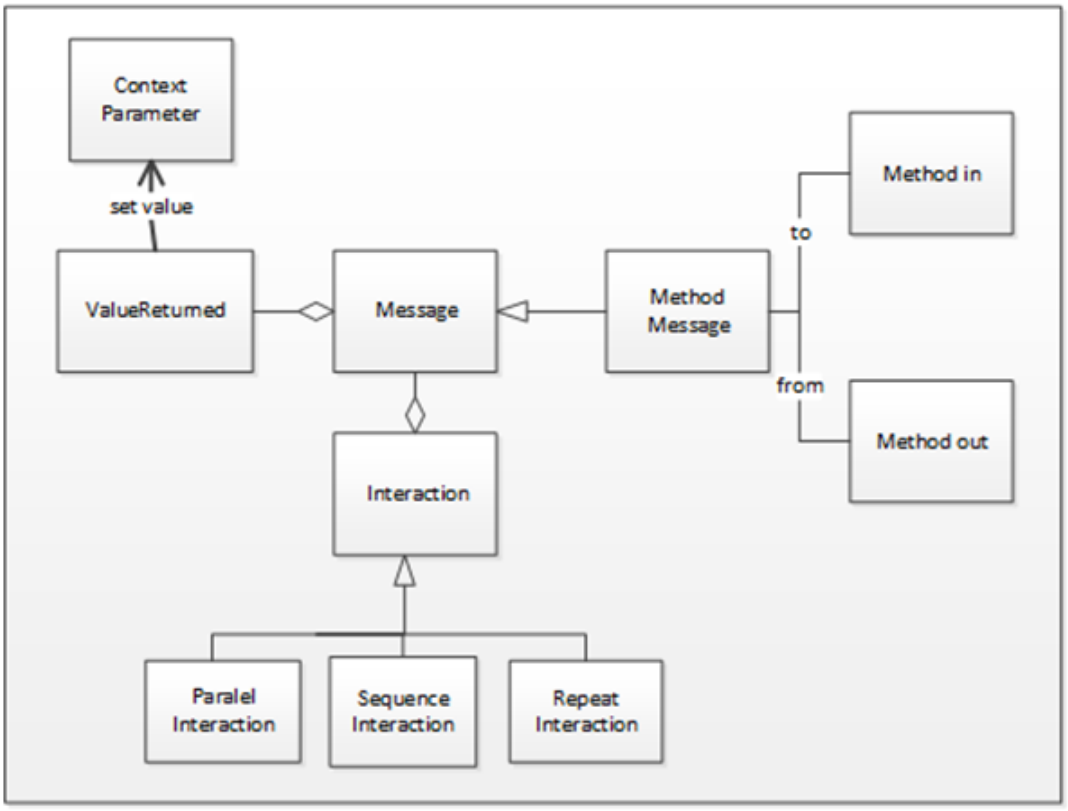

Figure 3.26: Composition Specification Constructs of XCOSEML Metamodel.

```
CompositionSpecification:
      "Composition"  name=ID
            (vconfmodelimport = VConfModelImport)?
            (cimports += ComponentImport)+
            ("Context Parameters" (contexts += ContextParameter)*)?
            ("Variability Mapping" (mappings += VMMapping)*)?
            ("Method" func += [MethodIn] ":" comp += Composition)+
;
```

Figure 3.27: XCOSEML CompositionSpecification Representation in Xtext.

VConfModelImport: *VConfModelImport* is the import mechanism to include configuration interface of the composition. Related grammar part is shown in Figure 3.28.

```
VConfModelImport:
      "import configuration" importedNamespace = ID
;
```

Figure 3.28: XCOSEML VConfModelImport Representation in Xtext.

ComponentImport: *ComponentImport* is the import mechanism to include sub components in component composition. Configuration interfaces of components can also be added with the *VarConfigurationModel* phrase. Grammar definition of *ComponentImport* is given in Figure 3.29.

```
ComponentImport:
      "has" s = [Interface]
      ( "with configuration"
            importedNamespace = [VarConfigurationModel])?
;
```

Figure 3.29: XCOSEML ComponentImport Representation in Xtext.

ValueReturned: This is an assignment of the return value of a method of an interface to a *ContextParameter* that is a global variable of composition. Related grammar definition can be seen in Figure 3.30.

```
ValueReturned:
      "%comp" name = [ContextParameter]
      "=" s = [Interface] "." m = [MethodIn] "%"
;
```

Figure 3.30: XCOSEML ValueReturned Representation in Xtext.

Composition: *Composition* is a set of interactions in order to realize a common goal by using one or more atomic (Message) or composite interactions (Interaction). Figure 3.31 shows the grammar definition.

```
Composition:
      (interactions += (Message | Interaction)
            (WS interactions += ( Message | Interaction))*)+
;
```

Figure 3.31: XCOSEML Composition Representation in Xtext.

3.3.3.1 Message Related Constructs

Message: *Message* is a basic interaction between two components with/without variability attachment (VariabilityAttachment). Also, a guard statement can be added to the message (IntConditionSet). For "send" and "receive" actions, "source" and "destination" interfaces are depicted. If the Message causes one or more changes in values of ContextParameters, then a set of ChorComputation is defined. Grammar definition is seen in Figure 3.32.

```
Message:
       (va = VariabilityAttachment)?
       ("guard(" guard = IntConditionSet")")?
       ( source =Source  type = "send"  "{"
              (destination += Destination )+ "} " |
      destination = Destination type = "receive"  ("from{"
       (sources += Source)+ "} "
       )? )
       (comp += ValueReturned)*
;
```

Figure 3.32: XCOSEML Message Representation in Xtext.

IntCondition: In Figure 3.33, *IntConditionSet* indicates a set of interaction conditions (IntCondition). A specification of a condition used to guard a part of an interaction. *IntCondition* can be either a definition of a condition with expression and numerical/non-numerical values or a specification of number.

```
IntConditionSet:
       icond += IntCondition (("or" | "and") icond += IntCondition )*
;

IntCondition:
       p1 = GUARDTEXT ((exp = EXPR (STRING | INT | ID | BOOLEAN) ) |
       "times")?
;
```

Figure 3.33: XCOSEML IntCondition Representation in Xtext.

Source: It consists the message's source interface and the method. Related component can be added to the beginning of the statement as depicted in Figure 3.34.

```
Source:
	(scompname = [Component] ".")?
	source = [Interface] "." mout=[MethodOut]
;
```

Figure 3.34: XCOSEML Source Representation in Xtext.

Destination: It consists of the message's destination interface and the method. Related component can be added to the beginning of the statement as depicted in Figure 3.35.

```
Destination:
	(dcompname = [Component] ".")?
	source = [Interface] "." min=[MethodIn]
;
```

Figure 3.35: XCOSEML Destination Representation in Xtext.

MethodIn and MethodOut: As shown in Figure 3.36, both forms of a method contain their names, their input parameters (*Params*, if there exist), and their output parameter (*Param*, if there exist).

```
MethodIn:
	"in" name = ID
	("input" ipars = Params)?
	("output" opar = Param)?
;

MethodOut:
	"out" name = ID
	("input" ipars = Params)?
	("output" opar = Param)?
;
```

Figure 3.36: XCOSEML MethodIn and MethodOut Representations in Xtext.

3.3.3.2 Interaction Related Constructs

Interaction: A definition of an interaction between components with/without a guard (IntCondition) includes; a set of repeating interactions (RepeatInt), a set of parallel

interactions (ParalelInt)and a set of sequential instructions (SequenceInt). Grammar definition of Interaction is in Figure 3.37.

```
Interaction:
	("guard(" guard = IntConditionSet ")" )?
	(interaction = RepeatInt |
	 interaction = ParalelInt |
	 interaction = SequenceInt
	)
;
```

Figure 3.37: XCOSEML Interaction Representation in Xtext.

RepeatInt: In Figure 3.38, *RepeatInt* defines a repetition of a set of interactions between components. These interactions can be atomic (Message) or composite (Interaction). There is an exit condition after the "repeat" keyword. A variability attachment can be added at the beginning. The interactions that will be repeated are placed between parentheses.

```
RepeatInt:
	(va = VariabilityAttachment)?
	"repeat" cond = IntConditionSet "("
		(interactions += (Message | Interaction))+ ")"
;
```

Figure 3.38: XCOSEML RepeatInt Representation in Xtext.

ParallelInt: A set of parallel interactions between components can be defined as in Figure 3.39. Interactions can be atomic (Message) or composite (Interaction). Variability attachment can be added before the "parallel" keyword. Interactions in the *ParallelInt* are surrounded with parentheses.

```
ParalelInt:
	(va = VariabilityAttachment)? "parallel ("
		(interactions += (Message | Interaction))+ ")"
;
```

Figure 3.39: XCOSEML ParalelInt Representation in Xtext.

SequenceInt: A definition of a sequence of a set of interactions between components

which can be atomic (Message) or composite (Interaction) with/without variability attachment. *SequenceInt* is written in such a way that the block is started with "sequence", interactions are surrounded with parantheses. Detailed grammar definition is given in Figure 3.40.

```
SequenceInt:
      (va = VariabilityAttachment)? "sequence ("
            (interactions += (Message | Interaction))+ ")"
;
```

Figure 3.40: I don't know how to reference to a figure.

3.3.4 Variability to COSEML Mapping Constructs

XCOSEML mapping constructs are shown in Figure 3.41.

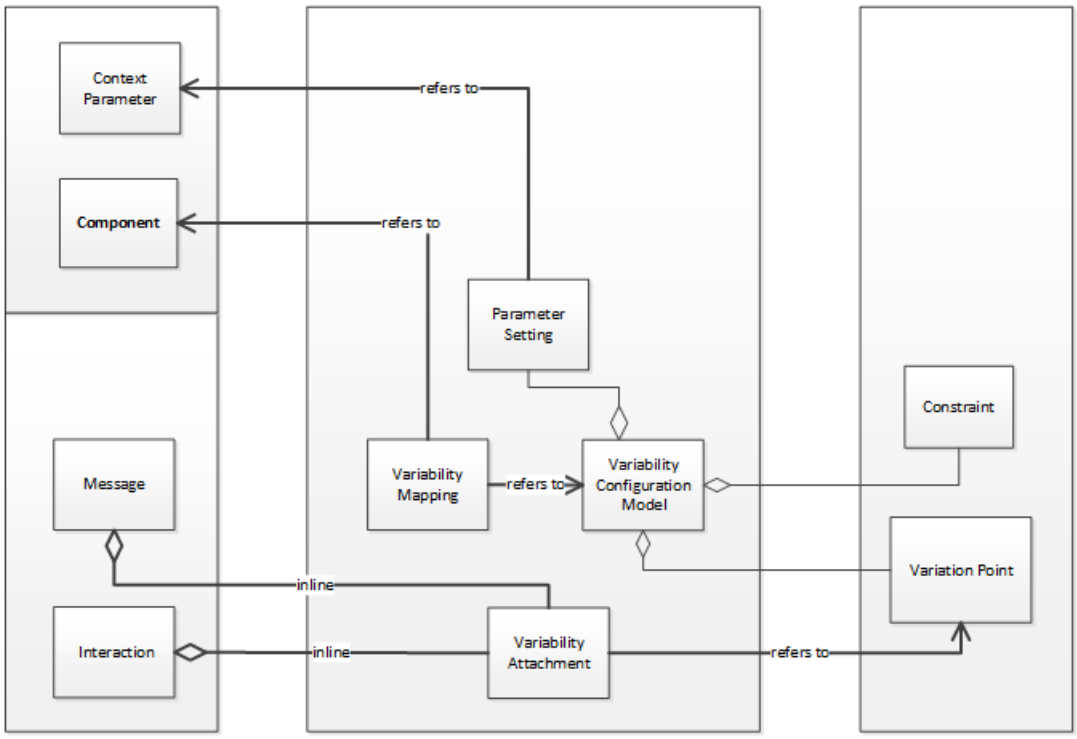

Figure 3.41: Variability to XCOSEML Mapping Constructs Metamodel.

VarConfigurationModel: *VarConfigurationModel* is a definition of a configuration interface whose grammar definition is given in Figure 3.42. Configuration interface

can belong to a component or a package. It contains; (i) a set of internal, external and configuration variation points (*VarPoint*) with a tag (*Tag*) defining the role of it if required, (ii) constraints (*Constraint*) among variation points and (iii) parameter settings (*ParameterSetting*) which includes a set of defined parameters used in component composition.

```
VarConfigurationModel:
       "Configuration interface" name = ID
      ("of component" compname = [Component] |
       "of package" packagename = [Package])
      ((tag += Tag)? vars += VarPoint)*
      ("Constraints" (constraints += Constraint )* )?
      ("Parameter Settings" (parametersetting += ParameterSetting )* )?
;
```

Figure 3.42: XCOSEML VarConfigurationModel Representation in Xtext.

ParameterSetting: *ParameterSetting* is an assignment of a value to "ContextElements" resided in composition specification. Figure 3.43 depicts the grammar definition.

```
ParameterSetting:
       "parameter" name = [ContextParameter]
      ("= #ofVariantsSelected{" (vars += [Variant])+ "} Of "
      vp = [VarPoint] |
       "= value(" var += [Variant] ("," vars += [Variant])* ")" |
      "existswhenselected{" vp=[VarPoint]"."v =[Variant]
       ("," vp2+=[VarPoint]"."v2 +=[Variant])* "}")
;
```

Figure 3.43: XCOSEML ParameterSetting Representation in Xtext.

VMMapping: *VMMapping* is a structural mapping from configuration variation to component variation. First variation points are mapped and then each variant of related configuration variation point is mapped to that of component variation point. The grammar definition is in Figure 3.44

```
VMMapping:
       "VP" vp = [VarPoint] ("maps component" | "maps package")
        element= [Interface] "VP" svp = [VarPoint]
       ("Variant"  vars += [Variant] "maps Variant"
              (mvars += [Variant] )+)+
;
```

Figure 3.44: XCOSEML VMMapping Representation in Xtext.

VariabilityAttachment: *VariabilityAttachment* is a definition of an attachment to composition specification in order to define the conditions of variation point and variant selections. Relationships between variation point and variants used are: "ifOneSelected" if one of the variants in a variant set is selected, "ifAllSelected" if all of the variants in a variant set is selected, "ifSelected" if some of the variants in a variant set is selected. Figure 3.45 shows the grammar definition for the variability attachment.

```
VariabilityAttachment:
       "#vp" vp += [VarPoint]
       ("ifOneSelected(" | "ifAllSelected(" | "ifSelected(")
       (vs += [Variant])+ (";excl:" (vsexc += [Variant])+)? ")"
       (("and" | "or") vp2 += [VarPoint]
      ("ifOneSelected(" | "ifAllSelected("| "ifSelected(")
      (vs2 += [Variant])+ (";excl:" (vsexc2 += [Variant])+)? ")" )* "#"
;
```

Figure 3.45: XCOSEML VariabilityAttachment Representation in Xtext.

3.4 Tool Support

XCOSEML files are of type package, component, interface, configuration interface and composition specification. We develop a tool in Java 1.8.0 to parse and configure the XCOSEML files. After parsing, the tool can configure the files based on the variant selections and produce related files as output. The XCOSEML parser reads these five types of files, categorizes the content of files, and saves them using purposively defined Java classes and provided data structures by Java. Another capability of the tool is to transform related XCOSEML files into their TVL and fPromela equivalent for model checking purposes. A visual description of the tool is given in Figure 3.46.

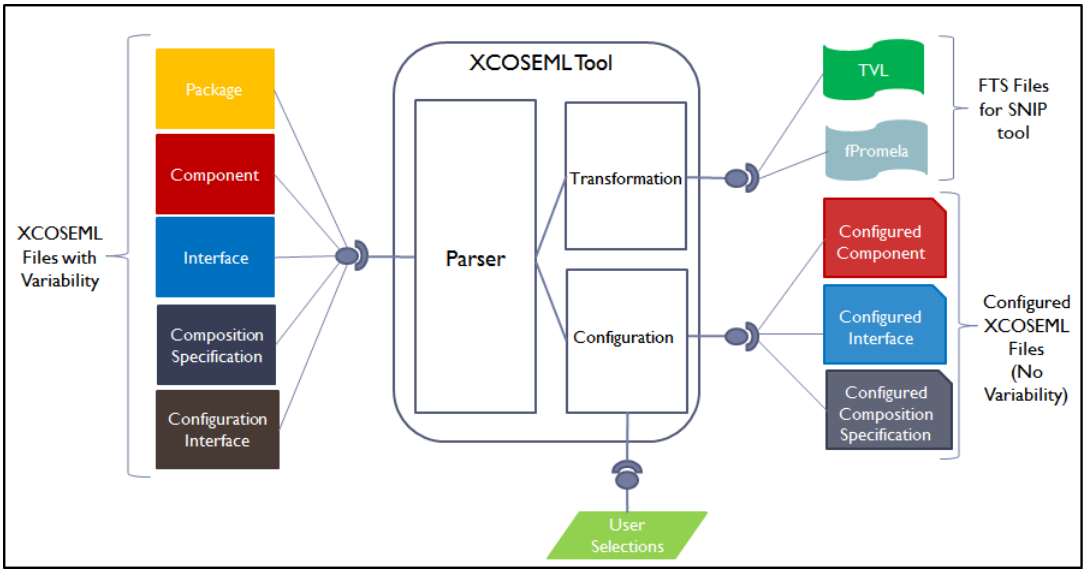

Figure 3.46: Visual Description of XCOSEML Tool.

Configuration of the system yields a customized model. For configuration, the tool is given a set of variants. Based on this set, the tool binds the variation points (VPs) in configuration interfaces. Then, if there exists, parameter settings are done depending on these bindings of VPs. Next, interfaces and composition specifications are tailored. In interfaces, functions and related parameters are selected or removed with respect to the chosen variants. In composition specifications, the parts which are guarded by VPs are added if the variants are selected. Then, parts that are related to variability are removed from composition.

In the case study given in section 3.1, configuration starts with the *smarthome_conf* configuration interface that belongs to the uppermost package *smarthome_pkg*. The full content of *smarthome_conf* can be found in Appendix B.1. There are two external VPs: *functionalities* at line 2 and configuration VP *homesize* at line 24. For *functionalities* VP, variants *security* and *safety* are mandatory. That means, they must be included for every product. Then we choose *telecare* from the optional variants of VP. Then, only one configuration variant, '*small*' is chosen based on the rule (min:1, max:1) at line 30 for the configuration VP '*homesize*'. With respect to the chosen configuration variant, one of the given variants of the VP '*bedroom*' is chosen (line 32). We select the variant '*onebedroom*' for this example. At the end of the variant selections, constraints are checked if some variants include or exclude some others.

We don't have any constraints in the *smarthome_conf*. Therefore, we have finished the configuration of *smarthome_conf*. In the following steps, configuration interfaces for member packages or components are configured. We have configuration interfaces for components *safety_cmp* and *security_cmp*; namely *safety_conf* and *security_conf*, respectively. External VPs are configured for these configuration interfaces while taking constraints into consideration. An excerpt from *safety_conf* configuration interface is given in Table 3.3.

Table 3.3: An excerpt from safety_conf.

```
1  Configuration interface safety_conf of component safety
2    internalVP deviceManagement:
3      optional
4        variant washingMachineController:activateMethods(
             turnOffDevices)
5        variant heatingCoolingController:activateMethods(
             turnOffHeating)
6        ...
7        variant ovenController:activateMethods(
             steepCoffeeAndTea)
8        ...
9        variant lightingController:activateMethods(
             turnOffLights,turnOffDevices)
10     bindingTime devtime
11
12   configuration home_type:
13     varType externalVP
14     alternative
15       variant apartment
16       ...
17     confvariant apartment mapping
18       VPName deviceManagement selectedVariants(
             washingMachineController heatingCoolingController
              generalElectricityController
             windowCurtainController ovenController
             cameraSystemController; min:1, max:6)
19       VPName sensors selectedVariants(voiceSensor gasSensor
              humiditySensor motionSensor tempratureSensor
             vibrationSensor waterleaksSensor; min:0, max:7)
20       ...
21   Constraints
22     ...
23     disasterDetection gasEmissionDetection requires
         sensors selectedVariants(gasSensor)
24     deviceManagement requires sensors selectedVariants(
         motionSensor voiceSensor)
```

After variant choice in configuration interfaces is finalized, related selections are done for composition specifications and interfaces based on the selected variant set. As a continuation of the example above, we configure *smarthome_comp* composition spec-

ification whose full content is given in Appendix C.1. For methods, variability attachments are checked; if corresponding variants are in the set of selected variants, then the interaction or message is a part of the composition. Otherwise, the interaction is removed. Also, all variability attachments are removed from the chosen interactions. After configuration of methods, imported components and context parameters are checked. If they are not used in the remaining interactions, they also removed. An excerpt from a configuration of *smarthome_comp* is given in Table 3.4. Full content of this configuration is given in Appendix D.

Table 3.4: An excerpt from configured smarthome_comp.

```
Composition smarthome_comp
  has security
  has entertainment
  ...

  Context Parameters
    residentdistance 0
    status "noproblem"

  Method onthewayhome:
    parallel (
      safety.steepCoffeeAndTea receive from{homeresident.
          cominghome}
      homeresident.getExerciseProgram send{ehealth.
          prepareExerciseProgram}
      homeresident.getAccessHistory send{security.
          reportAccessHistory}
      ...
    )
  Method goingOutside:
    parallel (
      security.lock receive from{homeresident.goingout}
      safety.turnOffDevices receive from{homeresident.
          goingout}
    )
  Method fallingDetection:
    sequence (
      repeat status != "fall" (
        ehealth.detectFalling receive from{homeresident.
            fell}
      )
      parallel (
        guard(status==spring) ehealth.
            getSurvelliancePictures send{security.
            captureSurvelliancePictures}
        ehealth.getAnomalyInfo send{safety.detectAnomaly}
          ...
    )
```

In configuration interfaces, a variant of a VP can indicate which methods are included

from the interface if that variant is chosen. After the name of the variant, a method name or a set of methods can be given with the keyword *activateMethods*. Therefore; if that variant is selected, then indicated method or methods must be included in the configured interface. An excerpt from *safety_conf* configuration interface is given in Table 3.3. Here activated methods are given with the variants. If variants *washingMachineController*, *heatingCoolingController*, *ovenController*, and *lightingController* are selected from *deviceManagement* VP, the *safety_int* interface is configured as in the Table 3.5.

Table 3.5: An excerpt from safety_int.

```
1  Interface safety
2    Provided Methods
3      in turnOffLights
4        input (datetime)
5
6      in turnOffHeating
7        input (datetime)
8
9      in steepCoffeeAndTea
10
11     in turnOffDevices
12       input (roomNumber)
13 ...
```

An interface belongs to a package or a component. Variability in components are defined in their interfaces. In other words, "variability in a component" is to select the needed methods from its interface. However, we are also handling component level variability. That is to choose some components in a system configuration and eliminate others. We are done with this configuration in composition configuration indirectly. We do not directly choose the desired components, though we eliminate the components (or the interfaces of those components) that are not to take place in the configured methods. Therefore, there is no separate configured file for component variability. The selected components are a part of the configured system.

The static view constructs of the configured system, namely packages, components

and interfaces, can be shown in a graphical representation to validate that it corresponds to a tree structure. Because our models are text-based, we should use a tool that transforms text-based input into a graphical representation. Structural decomposition of our case study is shown in Figure 3.1 by using the graphical COSEML tool. However, the current COSEML version does not take its input in a textual form. It is also not possible to resolve its saved *.cml* file since it is saved as Java stream. Therefore, temporarily we use Graphviz [26] for this purpose. Graphviz is open source and is used for graph visualization. It takes description of the graph in a simple text language. We adapt the textual version of static view constructs from the configured composition specification given in Appendix D. Then we transform these constructs into the language of Graphviz. The resulting tree structure is given in Figure 3.47

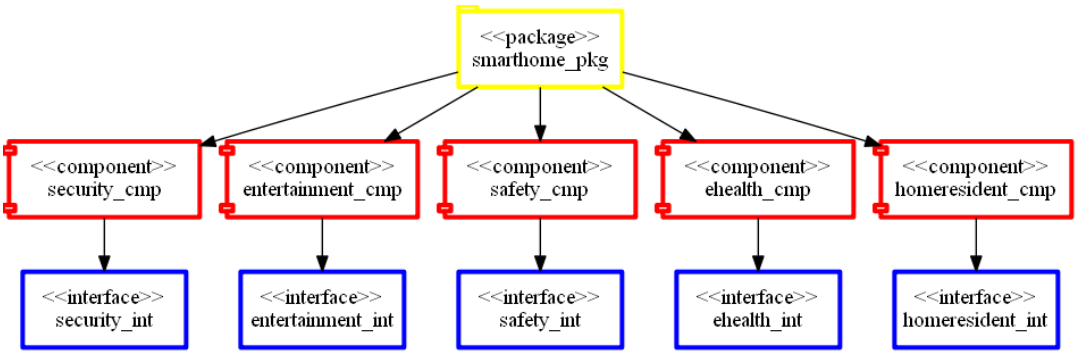

Figure 3.47: Configured Smart Home Model.

CHAPTER 4

VERIFICATION OF XCOSEML MODELS

In this chapter, our reason for using FTS's SNIP as a model checker is explained. XCOSEML to TVL and fPromela transformations are introduced. Partial automation of this transformation and constraints for fully transformation are discussed.

4.1 Why we choose FTS approach for model checking?

In [18], authors compare the FTS approach with the existing approaches in the literature, available as of the date of August, 2013. For instance, in [6] and [7], authors present an algorithm for model checking. This algorithm cannot be used to find the product that violates the system. In another research [42], there is a very similar modeling language with FTS' modeling language. However, their algorithm is much less efficient. SPLVerifier is a model checker described in [3]. The drawback of this model checker is it stops when it determines the first violation and it does not check the whole system.

In [53], authors verify their variable service orchestrations by using FTS approach. They have both variability model and behavior model, that is very similar to our approach. Their variability modeling technique is based on COVAMOF [49] and behavior model is BPEL [41]. Along with the ease of transformation of the models to be checked with the FTS's tools, they choose FTS because other approaches do not enable relating behaviours to products. However, FTS allows determining which products have violated the system.

Other reasons why we choose FTS are as follows:

- In the FTS approach, variability of system is represented with feature models and features are related with SPL behavior.
- The SNIP tool shows the violated products of the SPL together with its feature selections. This helps developers to find possible logical errors.
- Configuration interfaces of XCOSEML can be converted to feature models with little effort.
- Composition specifications of XCOSEML can be mapped to the behavior model of FTS' fPromela, even if one-to-one mapping cannot be achieved.

4.2 Transformation of Configuration Interfaces in XCOSEML to TVL Models

It is very common to represent variability in SPLs with feature models. In order to make models concise and brief, feature attributes should be used. However, intelligibility of semantics of feature models are decreased when using feature attributes. Most of the existing approaches do not support attributes. Also, graphical notation for feature model syntax seems as a drawback for industrial usage. TVL is a text-based feature modeling language designed to overcome the mentioned shortcomings. TVL provides a human-readable language with a rich syntax and formal semantics for designers. This makes modeling easy and models look natural [16].

TVL models in FTS include features, their attributes, and constraints in a hierarchical structure. In XCOSEML Configuration Interfaces there are variation points (internal, external and configuration), variants, constraints, and variation point associations. Because these two models are not semantically the same, we need a mapping between them.

In variability models for XCOSEML, all specified variation points are added to the root as mandatory. These variation points cannot be converted to features in TVL. Because the semantics between a set of variation points is hidden in choices of configuration, variation points that appear in configuration variation points must not be

added to the root directly in a feature model. Remaining variation points can be added to the root of the feature model directly.

The transformation steps from XCOSEML configuration variation point to TVL can be described as follows:

- Configuration interfaces that have composition specification are transformed to TVL.
- A root name is determined (usually the name of the configuration interface).
- The variation points of the configuration interface are checked: if they are not used in a configuration variation point, then they are added to the root directly. Otherwise, the variation point is not added because it will appear in the variants of the configuration variation point.
- Variation points are represented as the children of the root. Variants of the variation point are put in a block that is determined by curly braces ({}). The name of this block is the name of variation point. Variants are put in an inner block whose name indicates the type of them (e.g. mandatory, alternative or optional).
- Then, configuration variation points are added to the model. A main block is attached to the root with the name of the configuration variation point. Then configuration variants are put as inner blocks.
- Lastly, the constraints are converted to the TVL, if it is semantically possible.

4.3 Transformation of Composition Specifications of XCOSEML to fPromela

SNIP model checker employs *fPromela* as a procedural modeling language. fPromela is an extension of the language Promela that is the famous language of the SPIN model checker [36]. fPromela has the same syntax with Promela. Many constructs of these languages are similar to C. This makes it easy to get familiar with them. The key elements of the languages are *processes* that are specified with the *proctype* keyword. If a process is declared *active*, it is active in the initial state of the system. Otherwise, it has to be started by another process. If several processes are active at the same

time, they are processed alternately. The only way for processes to communicate is *channels*. Channels are defined globally. Local variables can be declared in processes.

A process' behavior is defined in a procedural style. Loops are declared with *do* statements. Loop bodies, i.e. options, are introduced each with a double colon (::). The first statement after the *do* keyword is the *condition* that the loop body is executed with respect to. We can also declare *if* statements in Promela and fPromela. These statements work similarly with *do* statements. They can have more than one option to execute. In this case, the language chooses which option to execute non-deterministically.

When processes communicate through channels, they write and read messages to the channel. The character "!" is used to write to a channel and "?" is used to read from a channel. Channel writes are done only if the channel is not full.

fPromela extends Promela with feature variables. Feature variables are declared as *bool* in a special structure namely *features*. This structure serves as an interface in which features are identified. Also, this structure ensures compatibility with Promela, as another benefit. Before referencing the features, a variable with this structure must be defined.

The variability in fPromela is expressed by *guarding* statements. These blocks are represented with the *gd* keyword. Guard statements has expressions similar to conditions of if statements. Features are used in these statements. The options of the guard statement are only part of the product if the featured expression in the guard holds.

We should transform XCOSEML composition specifications into fPromela equivalent. Therefore, we declare following transformation rules:

send: A send message in XCOSEML is defined as:

```
sourceInt.sourceMeth send{destinationInt.destinationMeth}
```

Firstly, a channel is created with the name of source and destination interfaces. Number of parameters of the methods is obtained. Then, based on number of parameters, this many *byte* keywords are added at the end of the channel definition. *byte* is a data type in TVL that holds one byte of information. For example if there are 2 param-

eters, the below channel will be defined. If the channel was created before, it is not created again.

```
chan chan_sourceInt.destinationInt=[1] of type {byte,byte}
```

The send message is transformed to fPromela as follows:

```
chan_sourceInt.destinationInt!parameter1,parameter2;
```

Here, *parameter1* and *parameter2* are put to the following channel:

```
chan_sourceInt.destinationInt
```

fPromela does not let the sending of a message without a parameter. If there is no defined parameter for methods in XCOSEML, then we pretend there is one parameter and put only one byte to the channel definition. Also, a fake variable is defined (e.g. temp) and put after the "!" as the input parameter:

```
chan chan_sourceInt.destinationInt=[1] of type {byte}
...
sourceInt.destinationInt!temp;
```

receive: A receive message in XCOSEML is defined as follows:

```
destinationInt.destinationMeth receive from{sourceInt.sourceMeth}
```

`from{...}` part is optional in XCOSEML. If this part is not used, it means that we don't care who the sender is.

For receive messages, a channel is created with the name of source and destination interfaces, as in send messages. Number of parameters is counted in order to add those many "byte"s at the end of the channel definition. If the channel was created before, it is not created again.

The receive message is transformed to fPromela as follows:

```
chan_sourceInt.destinationInt?parameter1,parameter2;
```

Here fPromela checks the channel for a message and puts this message to the given parameters.

In XCOSEML, we model the "receive a message" behaviour, hence we are not interested in the sender. Because we are defining an abstract definition for the message, there is no physical message. However, in fPromela, there should always be a message in the corresponding channel to avoid a deadlock. Therefore, if there is nothing in the channel, we should manually put something into the channel with a send message before the receive message appears. Again, if there is no specified parameters, a "temp" variable is placed after the "?".

parallel: This structure is shown in XCOSEML as:

```
parallel(exp1 exp2)
```

This structure can be transformed to fPromela as stated below:

```
fPromela-equivalent-of-exp1;fPromela-equivalent-of-exp2;
```

In fPromela, a parallel structure is employed by putting semicolon among expressions.

sequence: Sequence structure lets the expressions to be executed in order, from start to end. This expression appears in XCOSEML as given below:

```
sequence(exp1 exp2)
```

Transformation of the above expression into fPromela is as follows:

```
{fPromela-equivalent-of-exp1};
{fPromela-equivalent-of-exp2};
```

By putting expressions into curly braces and semicolons between them, we guarantee that the expressions are executed one after another.

repeat: In XCOSEML, repeat structure is used as follows:

```
repeat cond (exp1 exp2)
```

In this construct, *exp1* and *exp2* are executed until the *cond* condition does not hold. Corresponding fPromela structure is given below:

```
do
```

```
::  fPromela-equivalent-of-cond ->
fPromela-equivalent-of-exp1;
fPromela-equivalent-of-exp2;
::  else -> break;
od;
```

guard: A guard construct is shown in XCOSEML as follows:

```
guard(cond) exp1
```

fPromela structure of the given expression is as follows:

```
if
::  fPromela-equivalent-of-cond ->
fPromela-equivalent-of-exp1;
::  else -> break;
fi;
```

Variability Attachment: Variability attachments of XCOSEML are transformed to fPromela using *gd* statements.

In XCOSEML variability attachment is seen as:

```
#varAttachment# exp1
```

The fPromela transformation of the variability attachment is given below:

```
gd
::  fPromela-equivalent-of-varAttachment ->
fPromela-equivalent-of-exp1;
::  else -> break;
dg;
```

4.4 Verification of Smart Home Case Study

In this section we will explain how to verify our models of the case study introduced in section 3.1 after transformations. The SNIP tool can check the model against

deadlocks and assertions. A deadlock occurs, for example; when a send message tries to write to a channel while the channel is full, or a receive message tries to read an empty channel. The tool detects this problem by determining where its process gets stuck. It repeats itself waiting for a result; finally it produces an error indicating where the deadlock occurs. Assertions are not added to models at the transformation phase. Developers put assertions manually where they want to check. Then the models are checked against these properties. If there is a problem, the tool indicates that the assertion is violated. If there are no deadlocks and assertion violations, the process ends normally that means the model does not contain any problem.

The SNIP model checker uses TVL and fPromela files as explained before. It can be started using the following command line:

```
./snip -check -fm path/example.tvl path/example.pml
```

After the SNIP tool runs, it produces an output that shows results of the test. In the case of a deadlock or assertion violation, SNIP indicates where the problem occurs with line numbers and with an expression with features. For example, the model may work incorrectly in absence or presence of a future or some features. The tester can correct fPromela and TVL files directly. The correction can also be done in XCOSEML models that require a new transformation afterwards.

Variable package composition specifications are transformed by the tool. In other words, the composition specification which includes variability is transformed with its configuration interface. This condition is only assured by package configuration interfaces. Components cannot include subcomponents in our model, hence they do not need composition specification. In our case study *smarthome_conf* configuration interface is transformed to a TVL model based on the rules described in section 4.2.

Table 4.1: An excerpt from smarthome_conf configuration interface.

```
1  Configuration interface smarthome_conf of package smarthome
2    externalVP functionalities:
3      ...
4      optional
5        variant telecare
6        variant hobbygarden
7        ...
8      bindingTime devtime
9    internalVP bedroom:
10     optional
11       variant onebedroom
12       variant twobedroom
13       ...
14     bindingTime devtime
15   ...
16   configuration homesize:
17     varType externalVP
18     alternative
19       variant small
20       variant middlerange
21       variant big
22       (min:1,max:1)
23     confvariant small mapping
24       VPName bedroom selectedVariants(onebedroom twobedroom
             ; min:1, max:1)
25     ...
26     confvariant big mapping
27       VPName bedroom selectedVariants(threebedroom
             morethanthree; min:1, max:1)
28       ...
29     defaultVariant small
30     bindingTime devtime
```

Table 4.1 shows an excerpt from *smarthome_conf* configuration interface. There are three variation points (VPs): *functionalities* (line 2), *bedroom* (line 9), and *homesize* (line 16). While transforming models to TVL, *functionalities* VP is directly added to the root of the feature model. However, *bedroom* is not transformed directly since its configuration is done by the configuration VP *homesize*. The corresponding part of the transformed TVL file is given in Table 4.2. Variants of the *functionalities* VP

are grouped in blocks. For example, optional variants of the VP is grouped under the name *Fopt* (line 5). This name is generated by our tool by using the first letter of the VP (F) and the abbreviation of the variant type (opt). *group* and *someOf* are keywords of TVL. *someOf* indicates that the features in that block are "optional". Variants of the configuration variation points are given as blocks that contain their VP and variant mappings. For example, when *small* variant of the *homesize* configuration VP is selected, one of the variants of *bedroom* VP must be selected. This mapping can be seen at line 13. Here, the name of the block is the name of VP (*bedroom*). The numerical constraint is shown with the expression ([1..1]). The first integer indicates the minimum number of variants that can be selected, and the second one indicates the maximum number of variants.

Configuration variation points can configure a variation points in several ways. This requires repeating the name of the same variation point or variants more than once in the mappings of configuration VP variants. However, this is not a valid description in TVL syntax. Therefore, the transformation tool detects the repeated names and extends them with an underscore and an integer (e.g. _0) in order to get a different version. To indicate that they are names of the same entity, they are associated by using (<->) operator at the end of the TVL document. In our case, the name *Bedroom* is repeated two times. First occurrence is at line 13 and the second at line 19. The association is given at line 31. Full contents of *smarthome_conf* and its TVL transformation can be found in Appendix B.

Table 4.2: An excerpt from TVL equivalent of smarthome_conf.

```
1  root Smarthome_conf{
2          group someOf{
3                  Functionalities group allOf{
4              ...
5                          Fopt group someOf{
6                                  Telecare,
7                                  Hobbygarden,
8                                  ...
9                          }
10                 },
11                 Homesize group oneOf{
12                         Small group allOf{
13                                 Bedroom group [1..1]{
14                                         Onebedroom,
15                                         Twobedroom
16                                 }
17                         },
18                         Big group allOf{
19                                 Bedroom_0 group [1..1]{
20                                         Threebedroom,
21                                         Morethanthree
22                                 },
23                                 Outside group allOf{
24                                         Garden,
25                                         Garage
26                                 }
27                         },
28             ...
29                 }
30         }
31         Bedroom <-> Bedroom_0;
32     ...
33 }
```

Table 4.3 shows an excerpt from *smarthome_comp* composition specification and corresponding fPromela file part is given in Table 4.4. Here, the behaviours of three methods from the *smarthome* package are defined. In Table 4.3 at line 4, *steepCoffeeAndTea* method of *safety* interface receives a message from the *cominghome* method of the *homeresident* interface. The fPromela transformation of this part is at lines 3

and 4 in Table 4.4. Here the *temp* parameter is used in messages, because parameters are not defined in XCOSEML side, this is invalid for fPromela. We referred in Section 4.3 that messaging is done through channels in fPromela. The receive message is the first message of the method (XCOSEML), so it is the first message for proctype (fPromela). It means that corresponding channel is empty. Trying to read an empty channel results in a deadlock situation. To avoid that, the user must add a fake send message which writes something to the channel before the receive message reads it. Line 3 is added for this purpose. Similarly, transformation of a repeat interaction can cause a deadlock. In Table 4.3 in lines 7-9, a message is sent consistently while the value of *residentdistance* is smaller than or equal to 5. This means sending several messages to a channel in fPromela. That process ends up with a deadlock when the channel is full. Therefore, the user should add a fake receive message to avoid deadlock. The receive message at line 12 in Table 4.4 is included for this purpose. The full contents of the composition and fPromela files are in Appendix C.

Table 4.3: An excerpt from smarthome_comp compostion specification.

```
...
  Method onthewayhome:
    parallel (
      safety.steepCoffeeAndTea receive from{homeresident.
          cominghome}
      ...
      #vp functionalities ifSelected(telecare)# sequence (
        repeat residentdistance <= 5 (
          homeresident.sendDistance send{ehealth.
              getDistanceFromHome}
        )
        ...
      )
    )

  Method goingOutside:
    parallel (
      ...
      #vp functionalities ifSelected(entertainment)#
          sequence (
        ...
        entertainment.sendOutsideInfo send{ehealth.
            recommendOutfit}
      )
      ...

    )

  Method fallingDetection:
    #vp functionalities ifSelected(telecare)# sequence (
      repeat status != "fall" (
        ehealth.detectFalling receive from{homeresident.
            fell}
      )
      ...
      ehealth.setEmergentStatus send{security.
          notifyEmergency}
    )
```

Table 4.4: An excerpt from TVL equivalent of smarthome_comp.

```
...
proctype onthewayhome() {
    chan_safety_homeresident!temp;
    chan_safety_homeresident?temp;
    ...
    gd
    :: f.Telecare -> temp = temp + 1;
      {
        do
        :: cp_residentdistance <= 5 ->
          chan_homeresident_ehealth!personID;
          chan_homeresident_ehealth?cp_residentdistance;
        ::else -> break;
        od;
      };
      ...
    ::else -> skip;
    dg;
}
proctype goingOutside() {
    ...
    gd
    :: f.Entertainment -> temp = temp + 1;
      ...
      {
        chan_entertainment_ehealth!outsideTemperature;
      };
    ::else -> skip;
    dg;
    ...
}
proctype fallingDetection() {
  gd
  :: f.Telecare -> temp = temp + 1;
    ...
    {
      chan_ehealth_security!possiblereasons;
    };
  ::else -> skip;
  dg;
}
```

In Appendix B.2 and C.2, the tested TVL (smarthome.tvl) and fPromela (smarthome.pml) files are given, respectively. To run SNIP with these files, the following command is used:

```
./snip -check -fm smarthome.tvl smarthome.pml
```

After checking the models, SNIP ends up with the following output:

```
No never claim, checking only asserts and deadlocks..
Explored 701104, re-explored 347472, backtracked 1795184,
depth 35, buckets 701053, mean bucket size 1.00
Explored 1395940, re-explored 701212, backtracked 3595892,
depth 21, buckets 1395721, mean bucket size 1.00
No assertion violations or deadlocks found
explored 1743701 states, re-explored 888813.
```

This result shows that the tested smarthome models have no deadlocks or assertion violations. In order to experience a deadlock, the send message at line 72 in Appendix C.2 is removed. After running SNIP, the tool outputs the following messages:

```
No never claim, checking only asserts and deadlocks..
Found deadlock explored 123321 states, re-explored 62089.
-Products by which it is violated(as feature expression):
(Entertainment&Energymanagement&Hobbygarden&!Telecare)
```

This output indicates the deadlock and gives a feature expression that explains in which condition the deadlock occurs. SNIP can also trace the values of variables and shows them as output. This option can be disabled and only the products which cause the problem can be shown similar to the output above.

CHAPTER 5

CONCLUSION AND FUTURE WORK

5.1 Conclusion

In this thesis we describe a metamodel that includes hierarchical variability for Component Oriented Software Engineering (COSE) environment. We extend the COSE Modeling Language (COSEML) and call the new language as XCOSEML. We elaborate the language constructs and give a case study that is designed in XCOSEML. A tool is implemented in Java 1.8.0 to parse XCOSEML models and configure them for customization. The customized models are mapped to COSEML in order to make sure that they are valid. Model checking is done by using FTS methods. To check models in FTS's tool SNIP, model transformations are needed. These transformations are substantially done by the XCOSEML tool. However, there is still a need for tester's intervention of generated test codes to check the models properly.

When compared to the existing Component Based approaches that use variability, we have additionally covered different levels of a hierarchy, allowing binding of variability in a top-down approach. We also introduce an environment for verification of models with model checking which is the missing part for many approaches in the literature.

5.2 Future Work

Our experimentation with example models demonstrated the usability of our modeling approach. Yet industrial scale case studies are missing. Although there is an

underlying structure for variability of connectors, we haven't specified this yet. This will be done in the future.

We have a textual version of XCOSEML and a tool to parse and configure models. Although there is a graphical tool for COSEML, the ancestor of XCOSEML, we haven't extended this graphical tool with variability specifications.

For verification transformations, the tool is capable to transform models in some degree. As we discussed in the previous chapter a fully transformation is not possible because of semantic differences. However, some corrections and additions may be done in the automated model transformation tool as a future work.

REFERENCES

[1] Emad Albassam and Hassan Gomaa. Applying software product lines to multiplatform video games. In *Proceedings of the 3rd International Workshop on Games and Software Engineering: Engineering Computer Games to Enable Positive, Progressive Change*, pages 1–7. IEEE Press, 2013.

[2] Firas Alhalabi, Mathieu Maranzana, and J-L Sourrouille. A UML based methodology to ease the modeling of a set of related systems. In *Software Engineering Advances, 2008. ICSEA'08. The Third International Conference on*, pages 51–57. IEEE, 2008.

[3] S. Apel, H. Speidel, P. Wendler, A. von Rhein, and D. Beyer. Detection of feature interactions using feature-aware verification. In *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*, pages 372–375, Nov 2011.

[4] Timo Asikainen, Tomi Männistö, and Timo Soininen. Kumbang: A domain ontology for modelling variability in software product families. *Advanced Engineering Informatics*, 21(1):23 – 40, 2007.

[5] Timo Asikainen, Timo Soininen, and Tomi Männistö. A koala-based approach for modelling and deploying configurable software product families. In FrankJ. van der Linden, editor, *Software Product-Family Engineering*, volume 3014 of *Lecture Notes in Computer Science*, pages 225–249. Springer Berlin Heidelberg, 2004.

[6] Patrizia Asirelli, Maurice H. Ter Beek, Alessandro Fantechi, and Stefania Gnesi. A logical framework to deal with variability. In *Proceedings of the 8th International Conference on Integrated Formal Methods*, IFM'10, pages 43–58, Berlin, Heidelberg, 2010. Springer-Verlag.

[7] Patrizia Asirelli, Maurice H. ter Beek, Stefania Gnesi, and Alessandro Fantechi. A deontic logical framework for modelling product families. In David Benavides, Don S. Batory, and Paul Grünbacher, editors, *VaMoS*, volume 37 of *ICB-Research Report*, pages 37–44. Universität Duisburg-Essen, 2010.

[8] Felix Bachman and Paul C. Clements. Variability in software product lines. Technical Report CMU/SEI-2005-TR-012 ESC-TR-2005-012, Software Engineering Institute, Carnegie Mellon, September 2005.

[9] Heiko Behrens, Michael Clay, Sven Efftinge, Moritz Eysholdt, Peter Friese, Jan Köhnlein, Knut Wannheden, and Sebastian Zarnekow. Xtext user guide. *Dostupné z WWW: http://www. eclipse. org/Xtext/documentation/1_0_1/xtext. html*, 2008.

[10] Nelly Bencomo, Gordon S Blair, Carlos A Flores-Cortés, and Peter Sawyer. Reflective component-based technologies to support dynamic variability. In *VaMoS*, pages 141–150. Citeseer, 2008.

[11] David Blevins. Component-based software engineering. In George T. Heineman and William T. Councill, editors, *Component-based Software Engineering*, chapter Overview of the Enterprise Javabeans Component Model, pages 589–606. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[12] Barry W. Boehm. *Software Engineering Economics*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 1981.

[13] Jan Bosch, Gert Florijn, Danny Greefhorst, Juha Kuusela, J.Henk Obbink, and Klaus Pohl. Variability issues in software product lines. In Frank van der Linden, editor, *Software Product-Family Engineering*, volume 2290 of *Lecture Notes in Computer Science*, pages 13–21. Springer Berlin Heidelberg, 2002.

[14] John Cheesman and John Daniels. *UML Components: A Simple Process for Specifying Component-based Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[15] Andreas Classen. *Modelling and Model Checking Variability-Intensive Systems*. PhD thesis, PReCISE Research Centre, Faculty of Computer Science, University of Namur (FUNDP), 5000 Namur, Belgium, October 2011.

[16] Andreas Classen, Quentin Boucher, and Patrick Heymans. A text-based approach to feature modelling: Syntax and semantics of tvl. *Sci. Comput. Program.*, 76(12):1130–1143, December 2011.

[17] Andreas Classen, Maxime Cordy, Patrick Heymans, Axel Legay, and Pierre-Yves Schobbens. Model checking software product lines with SNIP. *International Journal on Software Tools for Technology Transfer (STTT), Springer-Verlag*, 14(5):589–612, 2012. DOI 10.1007/s10009-012-0234-1.

[18] Andreas Classen, Maxime Cordy, Pierre-Yves Schobbens, Patrick Heymans, Axel Legay, and Jean-François Raskin. Featured transition systems: Foundations for verifying variability-intensive systems and their application to LTL model checking. *IEEE Trans Software Eng (TSE)*, 39(8):1069–1089, 2013.

[19] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. The SEI series in software engineering. Addison-Wesley, 2002.

[20] Paul C. Clements. From subroutines to subsystems: Component-based software development, 1995.

[21] Itana Maria de Souza Gimenes, Fabrício Ricardo Lazilha, Edson Alves de Oliveira Junior, and Leonor Barroca. A component-based product line for workflow management systems. *CLEI electronic journal*, 7(2), 2004.

[22] A.H. Dogru and M.M. Tanik. A process model for component-oriented software engineering. *IEEE Software*, 20(2):34–41, Mar 2003.

[23] Ali H. Dogru. Component oriented software engineering modeling language: COSEML. Technical Report TR-99-3, Computer Engineering Department, Middle East Technical University, December 1999.

[24] Desmond F. D'Souza and Alan Cameron Wills. *Objects, Components, and Frameworks with UML: The Catalysis Approach*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[25] Eclipse. Xtext 2.7.0 available at. `https://eclipse.org/Xtext/`. Accessed: 2015-02-03.

[26] John Ellson, Emden Gansner, Lefteris Koutsofios, Stephen C North, and Gordon Woodhull. Graphviz—open source graph drawing tools. In *Graph Drawing*, pages 483–484. Springer, 2002.

[27] Tim Ewald. Component-based software engineering. In George T. Heineman and William T. Councill, editors, *Component-based Software Engineering*, chapter Overview of COM+, pages 573–588. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[28] Moritz Eysholdt and Heiko Behrens. Xtext: implement your language faster than the quick and dirty way. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pages 307–309. ACM, 2010.

[29] M. Galster, D. Weyns, D. Tofan, B. Michalik, and P. Avgeriou. Variability in software systems x2014;a systematic literature review. *Software Engineering, IEEE Transactions on*, 40(3):282–306, March 2014.

[30] Jerry Zayu Gao, Jacob Tsao, Ye Wu, and Taso H.-S. Jacob. *Testing and Quality Assurance for Component-Based Software*. Artech House, Inc., Norwood, MA, USA, 2003.

[31] Luca Gherardi. *Variability Modeling and Resolution in Component-based Robotics Systems*. PhD thesis, University of Bergamo, 2013.

[32] Amina Guendouz and Djamal Bennouar. Component-based specification of software product line architecture. In *International Conference on Advanced Aspects of Software Engineering, ICAASE*, pages 100–107, November 2014.

[33] A. Haber, H. Rendel, B. Rumpe, I. Schaefer, and F. van der Linden. Hierarchical variability modeling for software architectures. In *Software Product Line Conference (SPLC), 2011 15th International*, pages 150–159, Aug 2011.

[34] Mary Jean Harrold, Donglin Liang, and Saurabh Sinha. An approach to analyzing and testing component-based systems. In *First International ICSE Workshop on Testing Distributed Component-Based Systems, Los Angeles, CA*, pages 333–347. Citeseer, 1999.

[35] Peter Herzum and Oliver Sims. *Business Components Factory: A Comprehensive Overview of Component-Based Development for the Enterprise*. John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 2000.

[36] G.J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2004.

[37] M. Jahn, R. Rabiser, P. Grunbacher, M. Loberbauer, R. Wolfinger, and H. Mossenbock. Supporting model maintenance in component-based product lines. In *Software Architecture (WICSA) and European Conference on Software Architecture (ECSA), 2012 Joint Working IEEE/IFIP Conference on*, pages 21–30, Aug 2012.

[38] Kenneth Johnson, Radu Calinescu, and Shinji Kikuchi. An incremental verification framework for component-based software systems. In *Proceedings of the 16th International ACM Sigsoft symposium on Component-based software engineering*, pages 33–42. ACM, 2013.

[39] Muhammed Cagri Kaya, Selma Suloglu, and Ali H. Dogru. Variability modeling in component oriented software engineering. In *Proceedings of the 2014 Society for Design and Process Science*, 2014.

[40] Soo Dong Kim, Jin Sun Her, and Soo Ho Chang. A theoretical foundation of variability in component-based development. *Information and Software Technology*, 47(10):663 – 673, 2005.

[41] Michiel Koning, Chang-ai Sun, Marco Sinnema, and Paris Avgeriou. Vxbpel: Supporting variability for web services in bpel. *Inf. Softw. Technol.*, 51(2):258–269, February 2009.

[42] K. Lauenroth, K. Pohl, and S. Toehning. Model checking of domain artifacts in product line engineering. In *Automated Software Engineering, 2009. ASE '09. 24th IEEE/ACM International Conference on*, pages 269–280, Nov 2009.

[43] İbrahim İleri, Alperen Eroğlu, and Ali H. Dogru. Component-based variability modeling. In *Proceedings of the 2013 Society for Design and Process Science*, 2013.

[44] P. Oreizy, M.M. Gorlick, R.N. Taylor, D. Heimhigner, G. Johnson, N. Medvidovic, A. Quilici, D.S. Rosenblum, and A.L. Wolf. An architecture-based approach to self-adaptive software. *Intelligent Systems and their Applications, IEEE*, 14(3):54–62, May 1999.

[45] K. Pohl, G. Böckle, and F.J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 2005.

[46] Maryam Razavian and Ramtin Khosravi. Modeling variability in the component and connector view of architecture using uml. In *Proceedings of the 2008 IEEE/ACS International Conference on Computer Systems and Applications*, AICCSA '08, pages 801–809, Washington, DC, USA, 2008. IEEE Computer Society.

[47] R. Saidi, Nicolas Arnaud, D. Rieu, and M. Fredj. Multi-view variability modelling for business component reuse. In *Digital Information Management, 2007. ICDIM '07. 2nd International Conference on*, volume 2, pages 603–608, Oct 2007.

[48] Marco Sinnema and Sybren Deelstra. Classifying variability modeling techniques. *Inf. Softw. Technol.*, 49(7):717–739, July 2007.

[49] Marco Sinnema, Sybren Deelstra, Jos Nijhuis, and Jan Bosch. Covamof: A framework for modeling variability in software product families. In RobertL. Nord, editor, *Software Product Lines*, volume 3154 of *Lecture Notes in Computer Science*, pages 197–213. Springer Berlin Heidelberg, 2004.

[50] Neelam Sirohi and Anshu Parashar. Component based system and testing techniques. *International Journal of Advanced Research in Computer and Communication Engineering*, 2(6):2378–2383, June 2013.

[51] Selma Süloğlu. *Model-Driven Variability Management in Choreography Specification*. PhD thesis, Middle East Technical University, 2013.

[52] I. Sommerville. *Software Engineering*. International Computer Science Series. Pearson, 2011.

[53] Selma Suloglu, Riza Aktunc, and Mustafa Yucefaydalı. Verification of variable service orchestrations using model checking. In *Proceedings of the 2013 International Workshop on Quality Assurance for Service-based Applications*, QASBA 2013, pages 5–8, New York, NY, USA, 2013. ACM.

[54] Selma Suloglu, Bedir Tekinerdogan, and Ali H. Dogru. Choreography language for integration of variable orchestration specifications. In *Proceedings of Third International Symposium on Business Modeling and Software Design*, pages 121–130, 2013.

[55] Mikael Svahnberg, Jilles van Gurp, and Jan Bosch. A taxonomy of variability realization techniques: Research articles. *Softw. Pract. Exper.*, 35(8):705–754, July 2005.

[56] Maurice H. ter Beek and Erik P. de Vink. Software product line analysis with mcrl2. In *Proceedings of the 18th International Software Product Line Conference: Companion Volume for Workshops, Demonstrations and Tools - Volume 2*, SPLC '14, pages 78–85, New York, NY, USA, 2014. ACM.

[57] Tijs van der Storm. Variability and component composition. In Jan Bosch and Charles Krueger, editors, *Software Reuse: Methods, Techniques, and Tools*, volume 3107 of *Lecture Notes in Computer Science*, pages 157–166. Springer Berlin Heidelberg, 2004.

[58] J. van Gurp, J. Bosch, and M. Svahnberg. On the notion of variability in software product lines. In *Software Architecture, 2001. Proceedings. Working IEEE/IFIP Conference on*, pages 45–54, 2001.

[59] N. Wang, D.C. Schmidt, and C. O'Ryan. Overview of the CORBA component model. In George T. Heineman and William T. Councill, editors, *Overview of the CORBA Component Model*, pages 557–572. Addison-Wesley, 2001.

[60] Diana L. Webber and Hassan Gomaa. Modeling variability in software product lines with the variation point model. *Science of Computer Programming*, 53(3):305 – 331, 2004. Software Variability Management.

[61] Ye Wu, Dai Pan, and Mei-Hwa Chen. Techniques for testing component-based software. In *Engineering of Complex Computer Systems, 2001. Proceedings. Seventh IEEE International Conference on*, pages 222–232. IEEE, 2001.

[62] Gaoyan Xie. Decompositional verification of component-based systems-a hybrid approach. In *Automated Software Engineering, 2004. Proceedings. 19th International Conference on*, pages 414–417. IEEE, 2004.

APPENDIX A

XCOSEML GRAMMAR IN XTEXT

```
grammar org.xtext.xcoseml.XCoseml with org.eclipse.xtext.
    common.Terminals

generate xCoseml "http://www.xtext.org/xcoseml/XCoseml"

VarCompModel:
    (elements += AbstractElement)*
;

AbstractElement:
  CompositionSpecification | Interface |
      VarConfigurationModel | Package | Component
;

Component:
  "Component" name = QualifiedName
  ("Interface" (int = [Interface]))
  ("Configuration Interface" (confint = [
      VarConfigurationModel]))?
;

Package:
  "Package" name = QualifiedName
  ("includes" (element += [Construct])+ )
  ("Interface" (int = [Interface]))
  ("Configuration Interface" (confint = [
      VarConfigurationModel]))?
```

```
24    ("Composition" compname = [CompositionSpecification])?
25  ;
26
27
28  Construct:
29    Component | Package
30  ;
31
32  Tag:
33    "@" (name = "composition" | name ="vconfiguration" | name
          = "vconfrealization")
34  ;
35
36  VarConfigurationModel:
37    "Configuration interface" name = ID  ("of component"
          compname = [Component] | "of package" packagename = [
          Package])
38
39    ((tag += Tag)? vars += VarPoint)*
40
41    ("Constraints" (constraints += Constraint )* )?
42
43    ("Parameter Settings" (parametersetting +=
          ParameterSetting )* )?
44  ;
45
46  VarPoint:
47    ConfigurationVarPoint | InternalVarPoint |
          ExternalVarPoint
48  ;
49
50  InternalVarPoint:
51    vt = "internalVP" name=ID ':' variants = VariantSet  "
          bindingTime" btime =BINDING
52  ;
53
54  ExternalVarPoint:
```

```
55    ( vt = "externalVP" | vt2 = "vp")  name=ID ':' variants =
          VariantSet "bindingTime" btime =BINDING
56  ;
57
58  ConfigurationVarPoint returns ConfigurationVarPoint:
59    "configuration" ({InternalVarPoint} name=QualifiedName ':
          ' "varType"  vt = "internalVP" | {ExternalVarPoint}
          name=QualifiedName ':' "varType"  vt = "externalVP" )
60    (variants = VariantSet)
61    ("realization" rea = STRING) ((confvariants +=
          ConfVariantWithChoices)+ )
62    ("defaultVariant" defaultVariant = [Variant]) ("type"
          type= CONFTYPE "bindingTime" btime = BINDING )
63  ;
64
65  ConfVariantWithChoices:
66    "confvariant" name = ID "mapping"
67    (choices += Choice)+
68  ;
69
70  Choice:
71    "VPName" vp = [VarPoint] "selectedVariants(" (vars += [
          Variant])+  ("; min:" INT)? (", max:" INT)? ")"
72  ;
73
74  VariantSet:
75    {VariantSet} ("mandatory" (variants += Variant)* )?
76    ("optional " (variants += Variant)*)?
77    ("alternative " (variants += Variant)* "(min:" INT ",max:
          "INT ")")?
78  ;
79
80  Variant returns Variant :
81    "variant" name = ID  ( (":activateMethods(" funct = [
          MethodIn] ("," functs += [MethodIn])* ")")?
82      (":setParameter(toFunct:" f = [MethodIn] ",parameter:"
            pars = Param (";toFunct:" func += [MethodIn] ",
```

```
        parameter:" fpars += Param)*")" )?
83    )
84  ;
85
86  Constraint:
87    LogicalConstraint | NumericalConstraint
88  ;
89
90  LogicalConstraint:
91    ( p1 = [VarPoint] (p2 = [Variant])? ) c =CONST  p3 = [
        VarPoint] ("selectedVariants("(vars += [Variant])+ ("
        , min:" INT)? (", max:" INT)? ")")?
92  ;
93
94  NumericalConstraint:
95    lhs=RHS "const" rhs = RHS exp = EXPR
96    (STRING |  "valueOf{" (vars += [Variant])* "}")
97  ;
98
99  RHS:
100   (vp2 = [VarPoint] v2 = [Variant])
101 ;
102
103 ParameterSetting:
104   "parameter" name = [ContextParameter] ("= #
        ofVariantsSelected{" (vars += [Variant])+ "} Of " vp
        = [VarPoint] |
105   "= value(" var += [Variant] ("," vars += [Variant])* ")"
        | "existswhenselected{" vp=[VarPoint]"."v =[Variant]
106   ("," vp2+=[VarPoint]"."v2 +=[Variant])* "}")
107 ;
108
109 Interface:
110   "Interface" name = ID
111   //((functions += Function)+)
112   ("Properties" (compproperties += CompProperty)*)?
113   ("Provided Methods" (methodsin += MethodIn)+)
```

```
114     ("Required Methods" (methodsout += MethodOut)*)?
115   ;
116
117   Function:
118     "function" name = ID
119     ("precondition" precond += ID)?
120     ("postcondition" postcond += ID)?
121     ("input" ipars = Params)?
122     ("output" opar = Param)?
123   ;
124
125   Method:
126     MethodIn | MethodOut
127   ;
128
129   MethodIn:
130     "in" name = ID
131     ("input" ipars = Params)?
132     ("output" opar = Param)?
133   ;
134
135   MethodOut:
136     "out" name = ID
137     ("input" ipars = Params)?
138     ("output" opar = Param)?
139   ;
140
141   CompProperty:
142     name = ID
143   ;
144
145   CompositionSpecification:
146     "Composition"  name=ID
147       (vconfmodelimport = VConfModelImport)?
148       (cimports += ComponentImport)+
149       ("Context Parameters" (contexts += ContextParameter)*)?
150       ("Variability Mapping" (mappings += VMMapping)*)?
```

```
151     ("Method" func += [MethodIn] ":" comp += Composition)+
152 ;
153
154 VariabilityAttachment:
155   "#vp" vp += [VarPoint] ("ifOneSelected(" | "ifAllSelected
        (" | "ifSelected(") (vs += [Variant])+ (";excl:" (
        vsexc += [Variant])+)? ")"
156         (("and" | "or") vp2 += [VarPoint] ("ifOneSelected("
            | "ifAllSelected("| "ifSelected(") (vs2 += [
            Variant])+ (";excl:" (vsexc2 += [Variant])+)? ")
            " )* "#"
157 ;
158
159 Composition:
160   (interactions += (Message | Interaction) (WS interactions
         += ( Message | Interaction))* )+
161 ;
162
163 Interaction:
164   ("guard(" guard = IntConditionSet ")" )? (interaction =
        RepeatInt | interaction = ParalelInt | interaction =
        SequenceInt)
165 ;
166
167 RepeatInt:
168   (va = VariabilityAttachment)? "repeat" cond =
        IntConditionSet "("(interactions += (Message |
        Interaction))+ ")"
169 ;
170
171 ParalelInt:
172   (va = VariabilityAttachment)? "parallel (" (interactions
        += (Message | Interaction))+ ")"
173 ;
174
175 SequenceInt:
176   (va = VariabilityAttachment)? "sequence (" (interactions
```

```
        += (Message | Interaction))+ ")"
177 ;
178
179 Message:
180   (va = VariabilityAttachment)?
181   ("guard(" guard = IntConditionSet")")?
182   ( source =Source  type = "send"  "{" (destination +=
          Destination )+ "} " |
183    destination = Destination type = "receive"  ("from{" (
           sources += Source)+ "} ")? )
184   (comp +=ValueReturned)*
185 ;
186
187 Source:
188   (scompname = [Component] ".")? source = [Interface] "."
          mout=[MethodOut]
189 ;
190
191 Destination:
192   (dcompname = [Component] ".")? source = [Interface] "."
          min=[MethodIn]
193 ;
194
195 IntConditionSet:
196   icond += IntCondition (("or" | "and") icond +=
          IntCondition )*
197 ;
198
199 IntCondition:
200   p1 = GUARDTEXT ((exp = EXPR (STRING | INT | ID | BOOLEAN)
          ) | "times")?
201 ;
202
203 VMMapping:
204   "VP" vp = [VarPoint] ("maps component" | "maps package")
          element= [Interface] "VP" svp = [VarPoint]
205   ( "Variant"  vars += [Variant] "maps Variant" (mvars += [
```

```
        Variant] )+)+
206 ;
207
208 VConfModelImport:
209   "import configuration" importedNamespace = ID
210   /*[VarConfigurationModel]    */
211 ;
212
213 ComponentImport:
214   "has" s = [Interface] ( "with configuration"
        importedNamespace = [VarConfigurationModel])?
215 ;
216
217 ValueReturned:
218   "%comp" name = [ContextParameter] "=" s = [Interface] "."
        m = [MethodIn] "%"
219 ;
220
221 ContextParameter:
222   name = QualifiedName (defaultvalue = INT | STRING | ID |
        BOOLEAN)
223 ;
224
225 Params:
226   pars = "(" p1 = Param ("," p2 += Param)* ")"
227 ;
228
229 Param:
230   name = ID
231 ;
232
233 QualifiedName:
234   ID ('_' ID)*;
235
236 GUARDTEXT:
237   ID | INT
238 ;
```

```
239
240 TEXT:
241   (ID | INT | ":" | "/")+
242 ;
243
244 CONST:
245   requires = 'requires'| excludes ='excludes' | implies = '
        implies' | negates = 'negates'
246 ;
247
248 enum NUMCONST:
249   const = "const"
250 ;
251 BOOLEAN:
252   "true" | "false"
253 ;
254
255 enum EXPR:
256   gt = ">" |gte = ">="|lt = "<"| lte = "<=" | equ = "=="|
        neq = "!=" | eq = "="
257 ;
258
259 enum CONFTYPE:
260   subs = "substitution" | para ="parameterization" | add ="
        addition"
261 ;
262
263 BINDING:
264   devt = "devtime" | derv = "derivation" | comp = "
        compilation" | link = "linking" |strt= "start-up" |
        runt ="runtime"
265 ;
```

APPENDIX B

TRANSFORMATION FROM CONFIGURATION INTERFACE TO TVL

B.1 Configuration Interface File: smarthome_conf

```
1  Configuration interface smarthome_conf of package smarthome
2    externalVP functionalities:
3      mandatory
4        variant security
5        variant safety
6      optional
7        variant telecare
8        variant hobbygarden
9        variant entertainment
10       variant energymanagement
11     bindingTime devtime
12   internalVP bedroom:
13     optional
14       variant onebedroom
15       variant twobedroom
16       variant threebedroom
17       variant morethanthree
18     bindingTime devtime
19   internalVP outside:
20     optional
21       variant garden
22       variant garage
23     bindingTime devtime
```

```
24  configuration homesize:
25    varType externalVP
26    alternative
27      variant small
28      variant middlerange
29      variant big
30      (min:1,max:1)
31    confvariant small mapping
32      VPName bedroom selectedVariants(onebedroom twobedroom
            ; min:1, max:1)
33    confvariant middlerange mapping
34      VPName bedroom selectedVariants(threebedroom)
35      VPName outside selectedVariants(garden garage; min:1,
            max:2)
36    confvariant big mapping
37      VPName bedroom selectedVariants(threebedroom
            morethanthree; min:1, max:1)
38      VPName outside selectedVariants(garden garage)
39    defaultVariant small
40    type substitution
41    bindingTime devtime
```

B.2 TVL equivalent of smarthome_conf

```
1  root Smarthome_conf{
2          group someOf{
3                  Functionalities group allOf{
4                          Fman group allOf{
5                                  Security,
6                                  Safety
7                          },
8                          Fopt group someOf{
9                                  Telecare,
10                                 Hobbygarden,
11                                 Entertainment,
12                                 Energymanagement
13                         }
```

```
14                  },
15                  Homesize group oneOf{
16                          Small group allOf{
17                                  Bedroom group [1..1]{
18                                          Onebedroom,
19                                          Twobedroom
20                                  }
21                          },
22                          Big group allOf{
23                                  Bedroom_0 group [1..1]{
24                                          Threebedroom,
25                                          Morethanthree
26                                  },
27                                  Outside group allOf{
28                                          Garden,
29                                          Garage
30                                  }
31                          },
32                          Middlerange group allOf{
33                                  Bedroom_1 group allOf{
34                                          Threebedroom_2
35                                  },
36                                  Outside_3 group [1..2]{
37                                          Garden_4,
38                                          Garage_5
39                                  }
40                          }
41                  }
42          }
43          Bedroom <-> Bedroom_0;
44          Bedroom <-> Bedroom_1;
45          Threebedroom <-> Threebedroom_2;
46          Outside <-> Outside_3;
47          Garden <-> Garden_4;
48          Garage <-> Garage_5;
49  }
```

APPENDIX C

TRANSFORMATION FROM COMPOSITION SPECIFICATION TO FPROMELA

C.1 Composition Specification File: smarthome_comp

```
1  Composition smarthome_comp
2
3    import configuration smarthome_conf
4
5    has security
6    has entertainment
7    has safety
8    has garden
9    has energymng
10   has ehealth
11   has homeresident
12
13   Context Parameters
14     residentdistance 0
15     weatherInfo "a"
16     todolist "a"
17     status "noproblem"
18     anomaly "a"
19     images "a"
20     possiblereasons "a"
21
22   Variability Mapping
23     VP homesize maps component security VP securityPackage
```

```
24        Variant small maps Variant basic
25        Variant middlerange maps Variant advanced
26        Variant big maps Variant extended
27      VP homesize maps component safety VP home_type
28        Variant small maps Variant apartment
29        Variant middlerange maps Variant apartment
30        Variant big maps Variant house
31
32    Method onthewayhome:
33      parallel (
34        safety.steepCoffeeAndTea receive from{homeresident.
              cominghome}
35        #vp functionalities ifSelected(telecare)#
              homeresident.getExerciseProgram send{ehealth.
              prepareExerciseProgram}
36        #vp functionalities ifSelected(entertainment)#
              entertainment.prepareFavoriteWatchingList receive
               from{homeresident.cominghome}
37        #vp functionalities ifSelected(energymanagement)#
              sequence (
38          energymng.checkEnergyStatus receive from{
                homeresident.cominghome}
39          energymng.determineEnergyStatus send{homeresident.
                getEnergyStatus}
40        )
41        homeresident.getAccessHistory send{security.
              reportAccessHistory}
42        #vp functionalities ifSelected(hobbygarden)# garden.
              prepareHobbyGarden receive from{homeresident.
              cominghome}
43        #vp functionalities ifSelected(telecare)# sequence (
44          repeat residentdistance <= 5 (
45            homeresident.sendDistance send{ehealth.
                  getDistanceFromHome}
46          )
47          parallel (
48            security.unlock receive from{ehealth.
```

```
                setResidentCloseness}
49            #vp outside ifSelected(garage)# security.
                unlockGarage receive from{ehealth.
                setResidentCloseness}
50            entertainment.playList receive from{ehealth.
                setResidentCloseness}
51          )
52        )
53      )
54
55    Method goingOutside:
56      parallel (
57        security.lock receive from{homeresident.goingout}
58        safety.turnOffDevices receive from{homeresident.
            goingout}
59        #vp functionalities ifSelected(entertainment)#
            sequence (
60          entertainment.checkweather receive from{
              homeresident.goingout}
61          entertainment.sendOutsideInfo send{ehealth.
              recommendOutfit}
62        )
63        #vp functionalities ifSelected(hobbygarden)# garden.
            suggestPlant receive from{homeresident.goingout}
64
65      )
66
67    Method fallingDetection:
68      #vp functionalities ifSelected(telecare)# sequence (
69        repeat status != "fall" (
70          ehealth.detectFalling receive from{homeresident.
              fell}
71        )
72        parallel (
73          guard(status==spring) ehealth.
              getSurvelliancePictures send{security.
              captureSurvelliancePictures}
```

```
74          ehealth.getAnomalyInfo send{safety.detectAnomaly}
75        )
76        ehealth.analyzeStatus receive from{safety.setAnomaly}
77        ehealth.setEmergentStatus send{security.
              notifyEmergency}
78      )
```

C.2 fPromela equivalent of smarthome_comp

```
1  chan chan_safety_homeresident = [1] of {byte};
2  chan chan_homeresident_ehealth = [2] of {byte};
3  chan chan_energymng_homeresident = [1] of {byte};
4  chan chan_homeresident_security = [1] of {byte};
5  chan chan_garden_homeresident = [1] of {byte};
6  chan chan_security_ehealth = [1] of {byte};
7  chan chan_entertainment_ehealth = [1] of {byte};
8  chan chan_security_homeresident = [1] of {byte};
9  chan chan_entertainment_homeresident = [1] of {byte};
10 chan chan_ehealth_homeresident = [1] of {byte};
11 chan chan_ehealth_security = [1] of {byte};
12 chan chan_ehealth_safety = [1] of {byte};
13
14 int a=0;
15 int cp_possiblereasons=0;
16 int cp_images=0;
17 int cp_residentdistance=0;
18 int cp_anomaly=0;
19 int cp_weatherInfo=0;
20 int cp_todolist=0;
21 int noproblem=1;
22 int cp_status=1;
23 byte temp;
24 byte datetime;
25 byte currentmood;
26 byte date;
27 byte personID;
28 byte roomNumber;
```

```
29 byte now;
30 byte outsideTemperature;
31 byte outtime;
32 int fall=2;
33 int spring=3;
34 byte possiblereasons;
35 byte images;
36 byte anomaly;
37
38 typedef features {
39   bool Security;
40   bool Safety;
41   bool Telecare;
42   bool Hobbygarden;
43   bool Entertainment;
44   bool Energymanagement;
45   bool A1;
46   bool A2;
47   bool Onebedroom;
48   bool Twobedroom;
49   bool Threebedroom;
50   bool Morethanthree;
51   bool Garden;
52   bool Garage
53 };
54 features f;
55
56 active proctype smarthome(){
57   { run onthewayhome() };
58   { run goingOutside() };
59   { run fallingDetection() };
60 }
61
62 proctype onthewayhome() {
63     chan_safety_homeresident!temp;
64     chan_safety_homeresident?temp;
65     gd
```

```
66      :: f.Telecare ->
67        chan_homeresident_ehealth!temp;
68      ::else -> skip;
69      dg;
70      gd
71      :: f.Entertainment ->
72        chan_entertainment_homeresident!temp;
73        chan_entertainment_homeresident?datetime;
74      ::else -> skip;
75      dg;
76      gd
77      :: f.Energymanagement -> temp = temp + 1;
78        {
79          chan_energymng_homeresident!temp;
80          chan_energymng_homeresident?datetime;
81        };
82        {   chan_energymng_homeresident!temp; };
83      ::else -> skip;
84      dg;
85      chan_homeresident_security!date;
86      gd
87      :: f.Hobbygarden ->
88        chan_garden_homeresident!temp;
89        chan_garden_homeresident?temp;
90      ::else -> skip;
91      dg;
92      gd
93      :: f.Telecare -> temp = temp + 1;
94        {
95          do
96          :: cp_residentdistance <= 5 ->
97            chan_homeresident_ehealth!personID;
98            chan_homeresident_ehealth?cp_residentdistance;
99          ::else -> break;
100         od;
101       };
102       {
```

```
103             chan_security_ehealth!temp;
104             chan_security_ehealth?temp;
105             gd
106             :: f.Garage ->
107               chan_security_ehealth!temp;
108               chan_security_ehealth?temp;
109             ::else -> skip;
110             dg;
111             chan_entertainment_ehealth!temp;
112             chan_entertainment_ehealth?temp;
113         };
114     ::else -> skip;
115     dg;
116 }
117 proctype goingOutside() {
118     chan_security_homeresident!temp;
119     chan_security_homeresident?temp;
120
121     chan_safety_homeresident!temp;
122     chan_safety_homeresident?roomNumber;
123     gd
124     :: f.Entertainment -> temp = temp + 1;
125       {
126         chan_entertainment_homeresident!temp;
127         chan_entertainment_homeresident?now;
128       };
129       {
130         chan_entertainment_ehealth!outsideTemperature;
131       };
132     ::else -> skip;
133     dg;
134     gd
135     :: f.Hobbygarden ->
136       chan_garden_homeresident!temp;
137       chan_garden_homeresident?temp;
138     ::else -> skip;
139     dg;
```

```
140
141 }
142 proctype fallingDetection() {
143   gd
144   :: f.Telecare -> temp = temp + 1;
145     {
146       do
147       :: cp_status != fall ->
148         chan_ehealth_homeresident!temp;
149         chan_ehealth_homeresident?temp;
150         chan_ehealth_homeresident!cp_status;
151         chan_ehealth_homeresident?cp_status;
152       ::else -> break;
153       od;
154     };
155     {
156         if
157         :: (cp_status==spring) ->
158           chan_ehealth_security!date;
159           chan_ehealth_security?date;
160         ::else -> skip;
161         fi;
162         chan_ehealth_safety!datetime;
163         chan_ehealth_safety!cp_anomaly;
164     };
165     {
166       chan_ehealth_safety?anomaly;
167       chan_ehealth_safety!cp_possiblereasons;
168     };
169     { chan_ehealth_security!possiblereasons;  };
170   ::else -> skip;
171   dg;
172 }
```

APPENDIX D

CONFIGURED COMPOSITION SPECIFICATION

```
1  Composition smarthome_comp
2
3    has security
4    has entertainment
5    has safety
6    has ehealth
7    has homeresident
8
9    Context Parameters
10     residentdistance 0
11     status "noproblem"
12
13   Method onthewayhome:
14     parallel (
15       safety.steepCoffeeAndTea receive from{homeresident.
             cominghome}
16       homeresident.getExerciseProgram send{ehealth.
             prepareExerciseProgram}
17       homeresident.getAccessHistory send{security.
             reportAccessHistory}
18       sequence (
19         repeat residentdistance <= 5 (
20           homeresident.sendDistance send{ehealth.
                 getDistanceFromHome}
21
22         )
23         parallel (
```

```
24            security.unlock receive from{ehealth.
                  setResidentCloseness}
25            entertainment.playList receive from{ehealth.
                  setResidentCloseness}
26          )
27        )
28      )
29
30    Method goingOutside:
31      parallel (
32        security.lock receive from{homeresident.goingout}
33        safety.turnOffDevices receive from{homeresident.
              goingout}
34      )
35
36    Method fallingDetection:
37      sequence (
38        repeat status != "fall" (
39          ehealth.detectFalling receive from{homeresident.
                fell}
40        )
41        parallel (
42          guard(status==spring) ehealth.
                getSurvelliancePictures send{security.
                captureSurvelliancePictures}
43          ehealth.getAnomalyInfo send{safety.detectAnomaly}
44        )
45        ehealth.analyzeStatus receive from{safety.setAnomaly}
46        ehealth.setEmergentStatus send{security.
              notifyEmergency}
47      )
```