VARIABLE CONNECTORS IN COMPONENT ORIENTED DEVELOPMENT

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

BY

ANIL ÇETINKAYA

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

JUNE 2017

Approval of the thesis:

VARIABLE CONNECTORS IN COMPONENT ORIENTED DEVELOPMENT

submitted by **ANIL ÇETINKAYA** 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. Halit Oğuztüzün
Computer Engineering Department, METU _____

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

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

Assist. Prof. Dr. Ebru Aydın Göl
Computer Engineering Department, METU _____

Assist. Prof. Dr. Gül Tokdemir
Computer Engineering Department, Çankaya University _____

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: ANIL ÇETINKAYA

Signature :

ABSTRACT

VARIABLE CONNECTORS IN COMPONENT ORIENTED DEVELOPMENT

Çetinkaya, Anıl
M.S., Department of Computer Engineering
Supervisor : Prof. Dr. Ali H. Doğru

June 2017, 71 pages

Variability is incorporated in component oriented software development especially in the connectors besides components, for efficient configuration of software products in this thesis. Components have been regarded as the main building blocks in the development of software, especially in component based approaches. Connectors, however, were also part of the solution but with not much of a responsibility when compared to components. When considered in a holistic approach to yield executable code starting with the commonalities and variabilities in a domain model, one can realize the importance of the connectors: A realistic integration can and should utilize connectors for the various connector responsibilities, recently studied in the literature. Thus the connector structures are proposed to take place in the component model of COSEML within the classification for their responsibilities. Assigning more responsibilities to connectors suggests the enhancement of their internal structures with respect to some configurability along variability modeling and handling the tasks expected from the connector as such classifications require. This research defines the configurable mechanisms in connectors for 1) variability management and 2) conducting the defined responsibilities that are more than merely providing a connection port. As a future result, connectors will be managed like components, having some functionality and corresponding executable code in them. A case study is presented for the demonstration of the functioning of the proposed connector.

Keywords: Component Oriented Software Engineering, Domain Specific Language, Metamodel, Variability Modeling, Software Connector, Process Modelling

ÖZ

BİLEŞEN YÖNELİMLİ GELİŞTİRMEDE DEĞİŞKEN BAĞLAYICILAR

Çetinkaya, Anıl
Yüksek Lisans, Bilgisayar Mühendisliği Bölümü
Tez Yöneticisi : Prof. Dr. Ali H. Doğru

Haziran 2017 , 71 sayfa

Bu tezde, bileşen odaklı yazılım geliştirmede yazılım ürünlerinin etkin bir şekilde konfigürasyonu için, özellikle bileşenler dışındaki bağlayıcılara değişkenlik dahil edilmiştir. Bileşenler, özellikle bileşen tabanlı yaklaşımlarda yazılım geliştirmedeki başlıca yapı taşları olarak görülmektedir. Bununla birlikte, çözümün bir parçası olan bağlayıcılar ise bileşenlere kıyasla fazla bir sorumluluk taşımamaktaydı. Bir alan modelinde ortak noktalar ve değişkenliklerden başlayarak çalıştırılabilir kod üretmek için bütünsel bir yaklaşım düşünüldüğünde bağlayıcıların önemi anlaşılabilir. Yakın zamanlarda literatürde çalışıldığı üzere, gerçekçi bir entegrasyon için bağlayıcılara çeşitli bağlayıcı sorumlulukları yüklenilerek yararlanılmalıdır. Bu sebeple, bağlayıcı yapılarının sorumluluklarına göre sınıflandırılarak COSEML bileşen modelinde yer alması önerilmiştir. Bağlayıcılara daha fazla sorumluluk atamak, değişkenliğin modellenmesi ve belirlenen sınıflandırmaların gerektirdiği görevleri yerine getirmenin yanısıra iç yapılarının bazı konfigürasyon özelliklerine göre geliştirilmesini gerektirmektedir.Bu araştırma bağlayıcılarda 1) değişkenlik yönetimi ve 2) sadece bir bağlantı kapısı sunmak dışında tanımlanmış sorumlulukları yerine getirmek için tanımlanan yapılandırılabilir mekanizmaları tanımlar. Gelecekte bağlayıcıların bilşenler gibi yönetilerek, bazı işlevleri ve bunlara karşılık gelen çalıştırılabilir kodları olacağı düşünülmektedir. Önerilen bağlayıcıların işleyişinin gösterilmesi için bir durum çalışması içerilmiştir.

Anahtar Kelimeler: Alana Özgü Dil, Bileşen Yönelimli Yazılım Mühendisliği, Değişkenlik Modelleme, Metamodel, Süreç Modelleme, Bağlayıcı

To my family

ACKNOWLEDGMENTS

I would like to thank one of the greatest personalities I have ever known, my supervisor Professor Ali H. Doğru for his constant support, patience and guidance throughout this research. I have learnt a lot from him and I still have a lot more to learn. I also want to thank Professor Halit Oğuztüzün for his interest at my work and guidance he has offered. I also would like to thank committee members, Professor Ahmet Coşar, Assistant Professor Ebru Aydın Göl and Assistant Professor Gül Tokdemir for their valuable suggestions and feedback.

This thesis would not been possible without the support, help and motivation from M. Çağrı Kaya. You are one of the kindest persons I have ever met. Your help is really appreciated. I am also grateful to Dr. Selma Süloğlu for all the guidance and motivation she has offered.

I would like to thank my fellow friends Alper Karamanlıoğlu and Mehmet Koça, who I began this journey together. We have shared a lot, I really hope that we would always be there for each other.

I am thankful to my friends, Ahmet Rifaioğlu, Tuğberk İşyapar and Hasan Acar for their technical and moral support when most needed. I am also really thankful to my dear friends, Alperen Dalkıran, Fatih Calip and Samet Sezek for all the great memories that we get to share. It is always a gg with you guys.

I want to thank Ali Özkahraman, Hasan Abughosh, İbrahim Kılıç, Uğur Dal, Önder Çağlar, Hakan Yılmaz, İlhan Yumer, Ozan Koçak, Barış Küçük, Murat Arslan and Adem Ayan for their friendship and moral support. Also, I would like to thank to my friends and colleagues from CENG, Gökhan Özsarı, Alperen Eroğlu, Mahdi Saeedi Nikoo, Arınç Elhan, Abdullah Doğan, Aybike Şimşek Dilbaz, Murat Öztürk, Ahmet Atakan and Hüsnü Yıldız for their support and friendship.

Last but not least, I want to express my sincerest gratitudes to my father Necdet Çetinkaya, my mother Aysel Çetinkaya and my dear sister Dilan Çetinkaya for their constant support and unconditional love throughout my life.

TABLE OF CONTENTS

LIST OF TABLES

TABLES

LIST OF FIGURES

FIGURES

LIST OF ABBREVIATIONS

ADL	Architectural Description Language
ANTLR	Another Tool for Language Recognition
API	Application Programming Interface
BPMN	Business Process Modelling Language
CBD	Component Based Development
CBSE	Component Based Software Engineering
COSE	Component Oriented Software Engineering
COSEML	Component Based Software Engineering Modeling Language
DSL	Domain Specific Language
DMC	Disaster Management Center
DMS	Disaster Management System
FTS	Featured Transition System
IDE	Integrated Development Environment
MDD	Model Driven Development
MoRE	Model Based Reconfiguration Engine
SPL	Software Product Line
SPLE	Software Product Line Engineering
TVL	Textual Variability Language
VP	Variation Point
XCOSEML	Extended Component Based Software Engineering Modeling Language

CHAPTER 1

INTRODUCTION

1.1 Background

Reuse is an important paradigm for software development. Reusing software assets reduces costs and time to market. Component technologies are in the set of development methods that aim reusing software components systematically. Component Based Software Engineering (CBSE) or Component Based Development (CBD) use generally pre-built components to have a complex software system. Similarly, Component Oriented Software Engineering (COSE) aims reuse of software components. Different from CBSE, COSE takes advantage of having component technology during the whole development process. It considers development of components as the last option and does not impose a development method for this purpose.

Another important asset for the component based methods and sofware architectures is connectors. While components represent the functionality of the software, connectors deal with communication concerns among components. Their main role is to manage interactions among components.

Running programs directly from models is another important topic. Model-Driven Development (MDD) helps obtaining executable code from a model. This requires interpreting the model and adding extra information to achieve runnable code. Moreover, more simpler approaches are available such as mapping model assets to executable assets.

1.2 Problem Statement

Systems composed of components should ideally not involve the multi-purpose components for system-specific coordination duties. Components should be general purpose, not system aware. Therefore, a lot of code development is required for integration, that is the 'glue' functionality. However, the sole purpose of CBD is to reuse rather than code writing whereas this integration requires specific code development. Connectors, on the other hand, are very appropriate structures to assume this duty that have not been exploited sufficiently in this direction. Connectors, accommodating the interaction functionalities, should include necessary code for it, preferably that is configurable through variability management operating on a pre-defined set of functionalities. This is because most of the interaction expectations should also be reusable. Such use of connectors will both support the separation of concerns principle and reuse, if managed effectively.

1.3 Approach

Based on the separation of concerns principle, component oriented development is equipped with a detailed connector definition, in this research. This definition contains service type and connector type. To manage the connector selection process, variability management for connectors are also considered. In the proposed varibility approach when a connector is used, some parts of it can be included and other parts can be ignored. Therefore, the approach provides configurable connectors. In the proposed approach, variability takes place in the process model. Based on the variant selection some interactions -that are conducted by connectors- are included in the final model and others are ignored. At the end of the customization, the main execution stream of the system is produced. With an effort to map the model items to pre-implemented executables, developers can run the process model directly. It is assumed that a process model is also provided with a component model. Currently such a process model is distributed in the "package" primitives of the Extended Component Oriented Software Engineering Modeling Language (XCOSEML). Future enhancements may interface an XCOSEML specification with a process model

- such as expressed in Business Process Modelling Language (BPMN) for example. In this view, an executable system is considered as a dynamic part that is responsible for the ordered invocation of component methods, and a static part that is basically a declaration of a set of connected components. To improve reusability of the connectors, they are equipped with configuration capability, that is compliant with variability modelling.

1.4 Contribution

Contribution of this thesis is twofold: Firstly, connector definition of XCOSEML is enriched, that previously was only an abstract asset. With the new definition of connectors, XCOSEML's variability model can be reflected on connectors that results in variable connectors. Secondly, XCOSEML's tool support is improved with the ability of matching the languages process model to executable code. This is a step towards the executable modeling of XCOSEML.

1.5 Outline of Thesis

In chapter 2, some background information is provided on software connectors, CBSE, COSE, COSEML and XCOSEML. Also, importance of the variability in component oriented systems is explained. Some approaches are mentioned about variability modelling. Finally, connector types and their provided services are explained.

Chapter 3 includes some information for connector variability and connector definition. Furthermore, detailed background is given for connector definition in XCOSEML along with its relation with used mechanisms to manage variability. At the end of this chapter, Disaster Management System (DMS) case study is demonstrated.

In chapter 4, performed process to get an executable XCOSEML composition model is shown. Some important steps are detailed such as developing a tool to parse a configured XCOSEML composition file and developed mechanisms for matching included abstractions to running the pre-built code. Finally, Disaster Management System (DMS) case study is demonstrated in detail with the class diagram of the system

and sequence diagrams for different scenarios based on the configured XCOSEML composition file.

CHAPTER 2

BACKGROUND

In this chapter, some background information is provided on component-based and component oriented approaches, variability and its importance in software systems. Then, some information is given about Extended Component Based Software Engineering Modeling Language (XCOSEML) and its predecessor, Component Based Software Engineering Modeling Language COSEML. Finally, connector definitions are provided along with their provided services and types.

2.1 Component Oriented Software Engineering and COSEML

2.1.1 Software Components

Components are defined as implemented code blocks for software building [34]. They are usually pre built. Components are intended to use as run-time connecting modules which can be composed and deployed according to a composition standard. In this way, components can be seen as structural pieces of a software system.

Components should be composable and deployable independently from other components that are selected to yield a software system. Also, provided services by a component and required services for a component's operations should be shown in an interface provided for each of the components. Since there can be more than one component for the same purpose with small differences, their provided and required services for their operation should be clear and well documented [32]. So, the users can select the most suitable option according to their needs.

2.1.2 Component Based Software Engineering

Component Based Software Engineering (CBSE) is a sub-discipline of software engineering whose main aim is to separate the functional parts of software systems to be reused in different systems. The first applications of CBSE started in 1990s with the aim of constructing software components to be used in several software systems. In CBSE, first, functionally separable software components are defined and created. Subsequently, previously constructed software components are used to build large-scale software systems instead of creating the whole system from scratch. Object-oriented software development also aims software reuse. However, component based software development and object-oriented software development are different for several reasons. First, object-oriented approach cannot be used for general software reuse purposes, since each object is defined specifically in detail. Therefore, integrating objects to different software systems requires redefining the objects based on the new system. In addition, one who wants to use the previously defined objects should get all the source code, understand and modify it according to his or her needs which requires tremendous work. As a result, reusability, distribution and marketing of software objects is infeasible [32]. On the contrary, software components can be created and classified in a more generic manner without requiring detailed information. Components can be used by their interfaces which defines their functionalities without the need of implementation details. Components are larger than objects in general, however their reusability capabilities are higher. As software systems are becoming larger and larger in today's world, the importance and need of CBSE is increasing as well. This approach supports fast and reliable development of large-scale complex systems via reusable components that work in a harmony.

There are several characteristics of CBSE. First, each component is defined by its interface which is a high-level abstraction independent from the implementation of the software component. Software engineers can change the implementation of the components without changing its interface thanks to this feature of components. Therefore, a modified component can still be used without changing other parts of the systems. Second, components have some standards which define how their interfaces are used and how they will communicate. These standards provide necessary

information about the integration of components to the other components. The latter characteristic is that software components depend on a middleware which deals with low-level communication problems in the integration part of the components. Lastly, each software component should have a mechanism to enable the modification of the component based the functional needs.

2.1.3 Component Oriented Software Engineering

Component Oriented Software Engineering (COSE) is another approach in software engineering which is solely based on component oriented software approach rather than component based approach. This approach was first proposed in [10] in 2003. CBSE approaches are usually based on the object-oriented approaches with one exception: In CBSE, components can be defined and represented similar to objects. However, using object oriented approach limits the capabilities of programmers to utilize the main features and advantages of component based software development approach. In this approach, all phases of software development need to be component compatible. Therefore, software developers' main effort is spent on construction of components rather than composition of them. As a result, the main difference between the component based and component oriented approaches can be summarized as follows: In component based approaches developers focus on development of the overall system by other means (such as object orientation) and allow importing of components to it, which makes using component based approach less efficient whereas, in component oriented approach developers do not use conceptions about any structures other than components [10]. To eliminate this problem, component oriented system considers each component as a building block of the software system and these systems are constructed using a component model.

In COSE, modelling process starts with structural decomposition of the existing system with the aim of identifying the components to be integrated. The software system at hand should be represented in both physical and logical levels. The general process model of a COSE is given in Figure 2.1. Specifications of the system is constructed by following two phases which are decomposition and definition of extracted modules. The outcome of these two phases is a connected set of abstract components. Once

the abstract components are identified, the target components that are needed for the system are positioned within the system. The target components are like the implementations of the abstract components. At the end, these components are developed and integrated to the software system to construct the large-scale system.

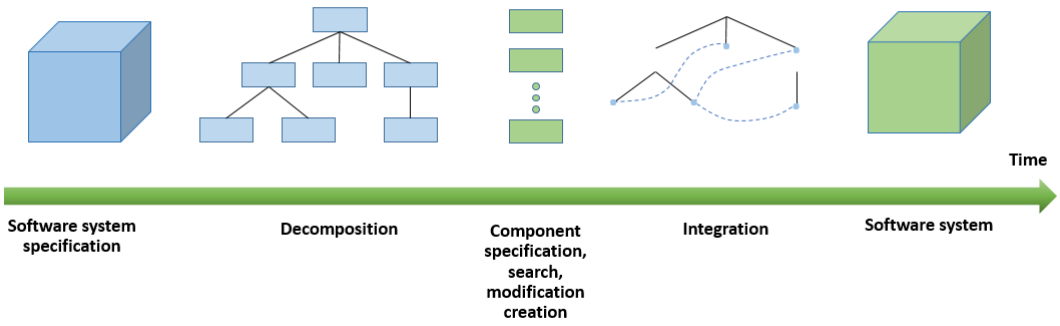

Figure 2.1: General development phases for component based development [10].

2.1.4 COSEML

A graphical modeling language is a language that provides a graphical representation of the components and their connections in a software system via set of consistent rules. COSEML stands for Component Oriented Software Engineering Modeling Language which is a graphical modeling language that was developed to be used in COSE [9]. COSEML has its own graphical tools for representation of components along with their connections. COSEML provides a way of developing software by composition rather than implementation which was explained in the previous section. This language provides the basic primitives, logical entities and implementation units. In COSEML, a development process starts with the identification and definition of parts of the system in an abstract level. In this first level, first, sub-systems should be determined. Subsequently, at the physical level, lower level system components should be defined. The system components defined in this level should include implementations of the abstract modules of the system which was determined in the previous level. There are several symbols in COSEML. For example, physical im-

plementation of an abstract module and corresponding abstract module are linked by the “represents” link. Connections among abstract entities and physical entities were described by a “connecter” symbol. COSEML includes five symbols to represent abstract entities which are “package”, “data”, “function”, “control” and “connector”, respectively. In physical level, the main symbol is “component”. Developers can also use special symbols to represent components having only one interface. If a component has more than one interface, these interfaces can also be represented by different symbols at the physical level where each symbol represents a unique interface. In this way, a system can be represented as a whole including its physical and logical components with structural and operational connections using the provided symbols in COSEML. The graphical symbols that can be used in COSEML is given Figure 2.2.

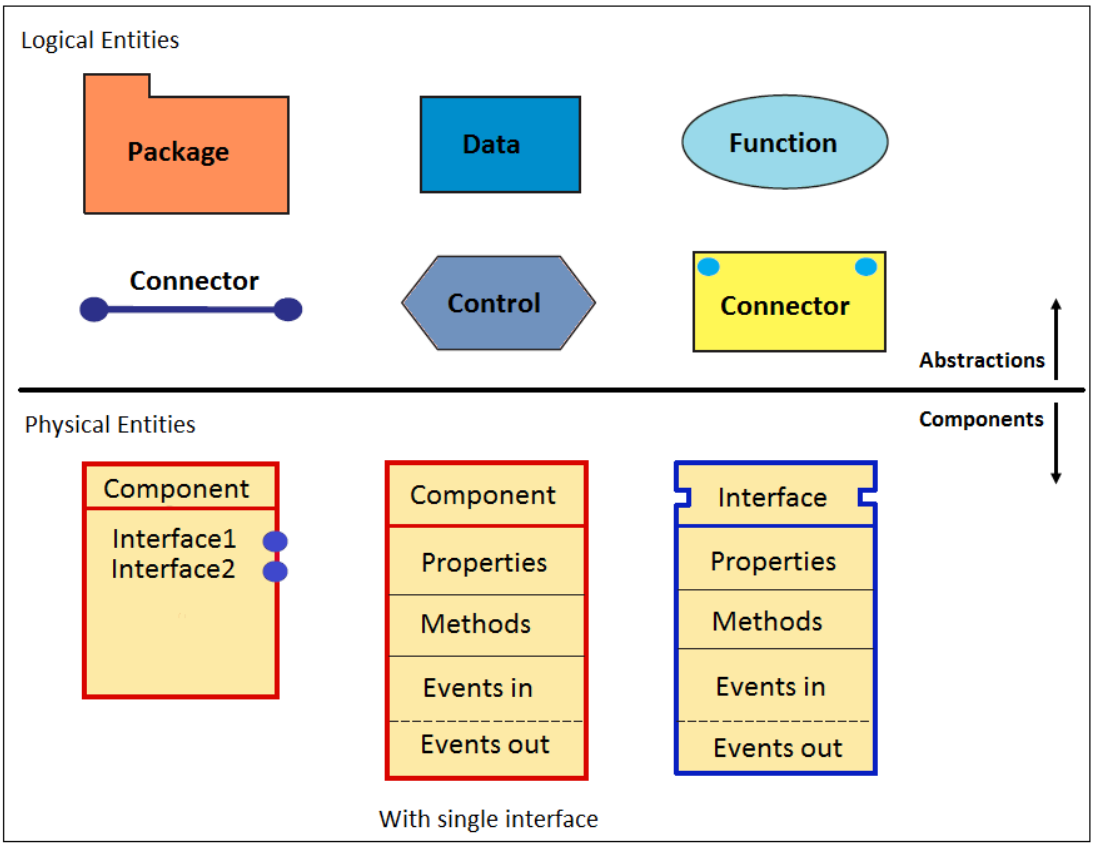

Figure 2.2: Graphical representations of COSEML assets [9].

2.2 Variability Modeling in Software Systems

2.2.1 Variability in Software Systems

As the software systems are getting bigger, complexity of the systems increases. In the old approaches, management of complex software systems was inefficient because any change or modification required to the system meant a change in the source code. Since it was very hard to keep track of all the interactions in the huge chunk of code, a change in the source code could create unexpected bugs or even crashes. With the introduction of CBSE, using pre-built components to develop software systems has become more and more popular rather than writing code from the scratch [10].

As a result of the increasing complexity, software systems required to be more dynamic and adaptable. In order to achieve this goal, software systems should have the ability to be changed, configured and extended and one of the ways to provide these abilities comes with the variability support [17].

Software Product Lines (SPL) are closely associated with variability management. They adopt a systematic approach to manage variability. In SPL, software assets should be created according to a defined software architecture and requirements of a software product family in order to be utilized later. Since it is possible for different products to share the same features and code, SPL can help to reduce development costs dramatically. Estimated development costs comparison between the systems developed with traditional methods and SPL approach is given in Figure 2.3 [28].

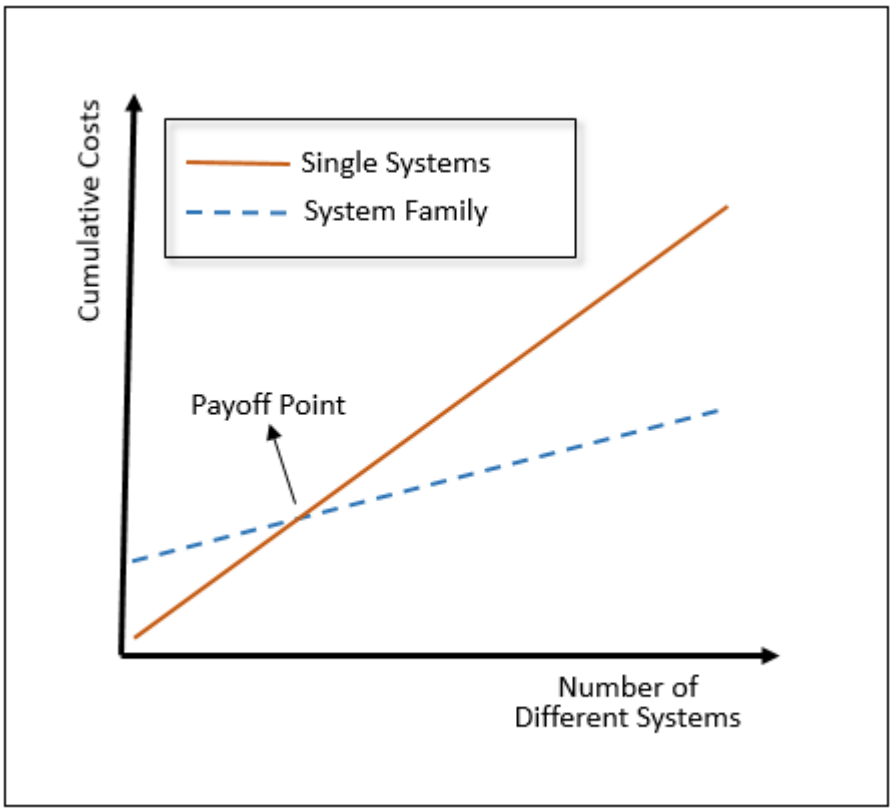

Figure 2.3: Development costs for n kinds of systems as single systems compared to SPL (adapted from [28]).

Delayed design principle is embraced in SPL as well as in run-time adaptive systems. There is no determined final product at the beginning. System will be determined along the development phase by selecting pre-defined alternatives or adding new modules to the system [25].

Figure 2.4. shows that there could possibly be infinitely many systems developed before seting any constraints. During the development some constraints are set and with every decision number of possible systems decreases. Each of these delayed design decisions causes system to differantiate and can be regarded as variation points. Finally, there could be only one configured system working at run-time. Delayed design decisions are usefull in SPL because it allows product line assets to be used more effectively according to changing requirements [36].

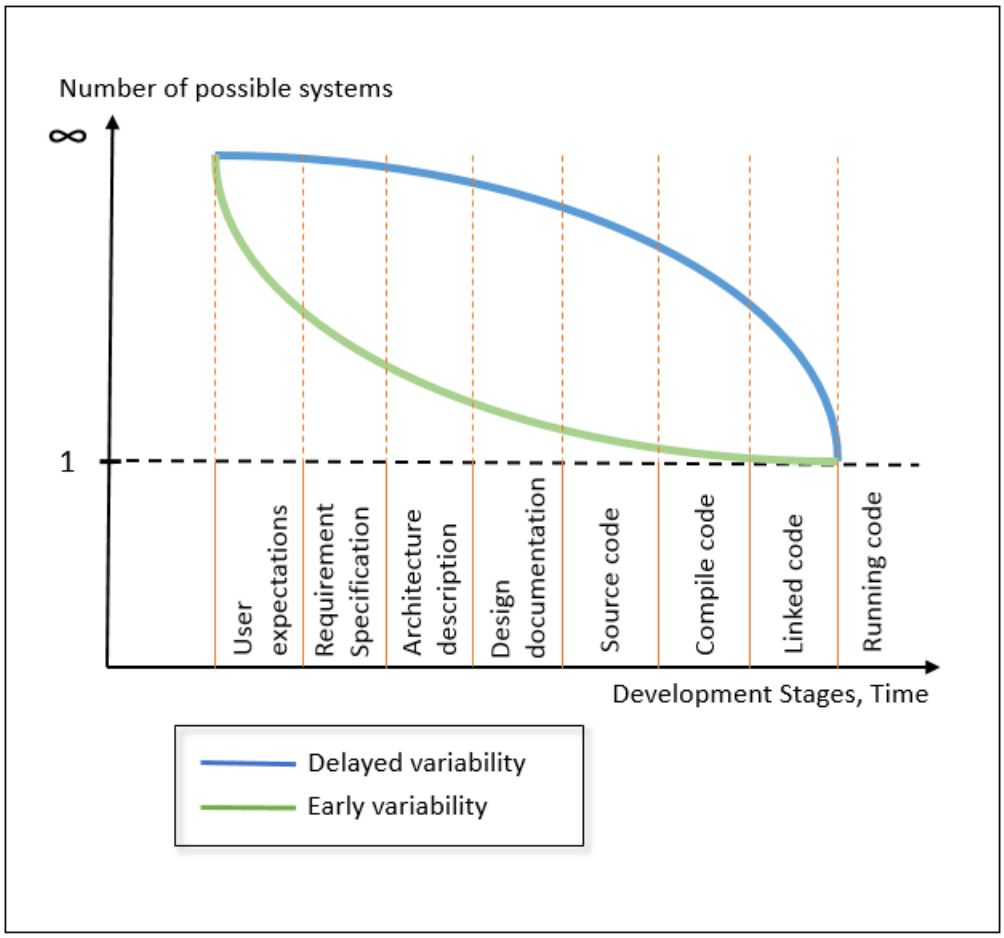

Figure 2.4: Variability change based on approaches(adapted from [36]).

2.2.2 Variability Modeling

Variability modeling is crucial to show variants of a software system efficiently in a formal way. Developing a system with variability allows developers to take advantage of reusability thus increase productivity. On the other hand, it may cause an increase in the complexity of the system. Therefore, systematic approaches are required to handle variability more effectively [2] [33] [3].

There have been several variability modeling approaches proposed to manage variability through all stages of the software development from requirements to running code. While these modeling approaches are introduced to achieve the same goal, they vary in model characteristics. These characteristics can be model choices, abstractions, tooling, guidance etc. A classification of proposed variability modeling

approaches is given in [31].While variability can be modeled in through all development phases, suitable modeling techniques may differ from phase to phase. For instance, feature models can be used in SPL when defining variability in requirements phase.

As it can be seen in Figure 2.4, variation points can be introduced in different levels of abstraction through all development stages ranging from requitements to running code. Also these variation points can be in implicit, designed and bound states based on following conditions. A variation point is implicit when it is shown in a higher level of abstraction. It is designed when design of a variability point took place in architectural design phase. A variation point is in binding state when it is bound to a variant in product architecture, during derivation time, compilation time, linking time, start-up time and run-time. [36]. Additionally, a variability point can either be open or closed due to its openness to adding new variants. If it is possible to add new variants it is open, closed if it is not.

2.2.3 XCOSEML

XCOSEML is a modeling language for the COSE approach. Its aim is to bring the advantages of component-orientation and variability management together. Unlike its predecessor COSEML it is a text-based language. Other than newly added connector definition, XCOSEML has five constructs: package, component, interface, configuration interface, and composition specification. Following paragraphs briefly introduce these concepts. XCOSEML packages correspond to the logical abstractions of the system. A package can represent a system as a whole, or a small part of the system (a subsystems) can be represented by packages. Packages are realized by physical entities, such as components. A package can include components, interfaces, connectors, and other packages. Components represent the physical equivalence of the system decomposition. They can be considered as running code of the system. They are usually pre-implemented. Their functionality is represented in their interfaces. Interfaces have provided and required methods. Provided methods represent the functionality that is conducted by the component itself. Each component must have at least one provided method. Required methods show the functionality that component needs

to be done outside of the component (i.e. in another component) to operate completely. Connectors were defined in XCOSEML metamodel when it was first defined. Although they took place in the metamodel as a construct, their definition was not provided in detail. In [5], XCOSEML is enriched with a detailed connector definition. Their definition, relationship with variability mechanism, and role in component communication are provided in Chapter 3. Configuration interface is the variability model for XCOSEML. It contains different types of variation points and constraints among them. Three types of variation points are defined; configuration, external, and internal variation points. Configuration variation points are high-level variation points that are used to configure low-level variation points. They are generally shown to developers to let them select desired variants. Then, they configure other variation points (lower-level ones) based on the variant selection. This is how variability binding occurs in XCOSEML hierarchically. External variation points are shown to the developers for variant selection. They are not abstract or high-level like configuration variation points. Configuration variation points are usually tagged as external to be open to developers' selection. Internal variation points are configured by other variation points to select specific functionalities of the system.

Composition specification is the process model of XCOSEML. It contains atomic and composite interactions. Atomic interactions are "send" and "receive" interactions conducted through connectors. Composite interactions contain "sequence", "parallel", and "repeat". Composite interactions contain atomic interactions and further composite interactions in a nested manner. Sequence contains a set of interactions to be executed consecutively. Parallel includes interactions to be executed concurrently. Repeat structure executes its content like a loop. Composition specification is a domain process model that contains all possible interactions with the defined system. It imports configuration interface - variability model of the language. Variation points and variants are associated to interactions through variability tags. Based on the variability bindings, corresponding interaction are included in the configured composition file. However, this customized XCOSEML process model is not executable yet. In Chapter 4, mapping XCOSEML constructs to executable assets is introduced. The aim here is to run the desired system directly from XCOSEML.

2.3 Software Connectors

Modern software systems consist of many complex components. Components are responsible for both processing and data, separately or together at the same time. The interactions among these components are one of the main issues in CBSE approaches. Management of these components can be challenging considering the size and complexity of modern systems. Interactions among components can be even more important than components themselves with regard to ensuring the system stability and extendibility from an architectural point of view.

A definition for connector is given as "architectural element tasked with effecting and regulating interactions among components" in [35]. According to this definition, connectors are abstractions in charge of the interactions among components in an architectural level.

Connectors have generally indicated themselves as simple procedure calls or shared data access in traditional software systems. In terms of architecture, they were treated as invisible. Abstractions were symbolized as lines and boxes where components were represented by boxes, connectors were represented by lines. These lines were simply inadequate to represent the identity or properties of a connector. Also, these connectors were only able to manage the interactions between pairs of components. Increasing complexity of software systems caused connectors to adapt and evolve. More demanding requirements have led connectors to have their own identities, roles and bodies of codes. They have also gained an ability to work with many different components, even simultaneously.

In simple terms, software connectors are responsible for transfer of the control and data among components. While the most of the components take on specific roles in the composition of an application, connectors are usually independent from application or context.

2.3.1 Classification of Connectors

There are four general classes of services that can be provided by a software connector; communication, coordination, conversion and facilitation[24].

Communication: These connectors are used for the transmission of data among components. In component interactions, data transmission services are one of the major building blocks. This type of connectors can be used for passing messages, exchanging data to be processed and communication of the computation results.

Coordination: These connectors are responsible for transfer of control among components. Interactions among components are handled by switching thread of execution from one component to another. Function calls and method invocations can be given as examples.

Conversion: Conversion connectors allows heterogeneous components to interact with each other. It is a major task considering that probable mismatches is one of the major problems in the way of building large and complex systems.

Facilitation: This kind of connectors are used for mediating and streamlining component interactions. Heterogeneous components needs some mechanisms even if they have been created to interoperable with each other. Facilitation components are very useful for facilitating and optimizing component interactions.

2.3.2 Commonly Used Connector Types

Abstractions obtained by these four classes are not enough to build connectors since they do not provide sufficient details to model and analyze them. Therefore, eight connector types are defined based on which way they realize their roles in interactions [24]. These connector types and their probable classes are shown in Table 2.1.

Procedure Call Connectors: This is the most widely used type of connectors. This type of connectors is used to manage the distribution of control and synchronous data between a pair of components. Hence, they can be classified as coordination and communication connectors. The caller component forwards the thread of control

Table 2.1: Services provided by connector types

		Services Provided			
		Communication	Coordination	Conversion	Facilitation
Connector Types	Procedure Call	X	X		
	Event	X	X		
	Data Access	X		X	
	Linkage				X
	Stream	X			
	Arbitrator		X		X
	Adaptor			X	
	Distributor				X

along with the parameters to the callee component; then the callee returns the control along with the results to the caller when the operation completed. Object oriented methods, fork and join in environments such as Unix, operating system calls can be given as examples of this types of connectors.

Event Connectors: Event connectors model the flow of control among components just like it was in the procedure call connectors, on the other hand unlike procedure call connectors the flow is disrupted with an event. Therefore, they provide coordination services.

Data Access Connectors: This type of connectors provides communication services to allow components to access a shared memory. Since data access can occur in distributed time, the component which data is stored should be invoked for preparation of data before the access and should clean up after the access. Data access connectors can also provide conversion services if there is a difference in the stored data format and required data format.

Linkage Connectors: Purpose of these connectors are to bundle the system components to each other and hold them together along their operation. By this way, they provide the channels for communication and coordination for higher order connectors to communicate through. Linkage connectors are used for facilitation of enforcing the interaction semantics. Main goal of this type of connectors are to assist repair, expand and monitor an existing system. After the foundation of channels in a system, linkage connectors can disappear or remain in the system in order to help with extendability.

Stream Connectors: Stream connectors are used for communication purposes by performing considerable amounts of transfers among autonomous processes. They can also be used for delivering computation results in client server systems with data transfer protocols. This type of connectors have been utilized to represent connectors with complex usage protocols in formal architectural models [1], [30]. Stream connectors can be paired with various connectors such as data access connectors to perform database and file storage access, or event connectors to help with the delivery of multiple events. UNIX pipes, TCP/UDP communication sockets can be given as examples of stream connectors.

Arbitrator Connectors: Presence of other components can effect the operation of a component when the states and and requirements are unknown in a software system. Arbitrator connectors are used for streamlining system operations and resolving conflicts. They can also redirect the control flow for coordination purposes. Multi threaded systems with shared memory access can be given as a usage area for these types of connectors.

Adaptor Connectors: Since component oriented systems are built from pre-implemented components, some inconsistency among components is almost inevitable. Adaptor connectors provide conversion services to ensure the interactions among heterogeneous components. Conversion can also be done to optimize the interactions in a software system. The systems include different computing platforms or programming languages can be given as examples which requires adapter connectors.

Distributor Connectors: These connectors exist to provide the assistance required by other connectors such as procedure calls and streams. They help to acknowledge the paths and allow routing through these paths in order to allow other components to communicate and coordinate more efficiently. They are classified as facilitation connectors.

CHAPTER 3

DEFINING VARIABLE CONNECTORS IN XCOSEML

In this chapter, extension of the Extended Component Based Software Engineering Modeling Language (XCOSEML) language with explicit connector definition and associating them with variability are introduced. In Section 3.1, a literature summary on connector definition and connector variability is provided. XCOSEML connector definition is explained in detail. Finally, the relationship between variability mechanism of the language and connector definition is explained.

3.1 Software Connectors and Variability

As first-class citizens of the software modeling, connectors should be taken into account while modelling variability. During the system development, or even at runtime, connectors should be added or removed to the system. Besides plugging a connector directly to a system, a connector is supposed to be configured for different purpose of usage. This depends on the definition of the connectors. A detailed and precise definition provides developers a chance to select desired parts, and discard undesired parts inside the connectors. There are some approaches that consider connector as a variable asset. In [29], connector variability is used in the component and connector view of software architecture. However, their connector definition is not detailed and connectors' operations are not explicitly defined. In [15], a mapping to software connectors are provided in their hierarchical variability modeling approach, again without detailed connector definition. In [14], an approach is proposed for integration of Software Product Line Engineering (SPLE) and Component Based Soft-

ware Engineering (CBSE). This approach also does not have a detailed connector definition. Other approaches that "contain variable connectors" can be classified as follows: Cetina et al. [4] propose Model-Based Reconfiguration Engine (MoRE) with the capability of run-time adaptability in the context of autonomic computing. Using different communication channels during reconfiguration is a method of defining variability to communication among components. Although this is a kind of connector variability, the variability logic is hidden in feature models that makes it difficult to manage it in large-scale systems. In [13], connector variability is conducted through ports of a specific component. In [16], authors employ connector variability through adaptation middleware. Dynamic reconfiguration is applied to components and their connections.

3.2 XCOSEML Metamodel and Connector Extension

XCOSEML adds dynamic constructs to static constructs of the Component Based Software Engineering Modeling Language (COSEML). Also, as an important extension, it extends COSEML with variability. For the sake of separation of concerns, variability constructs are separated from language constructs. For the connection of these two model parts, mapping constructs are used. Figure 3.1 depicts an overview of the XCOSEML metamodel.

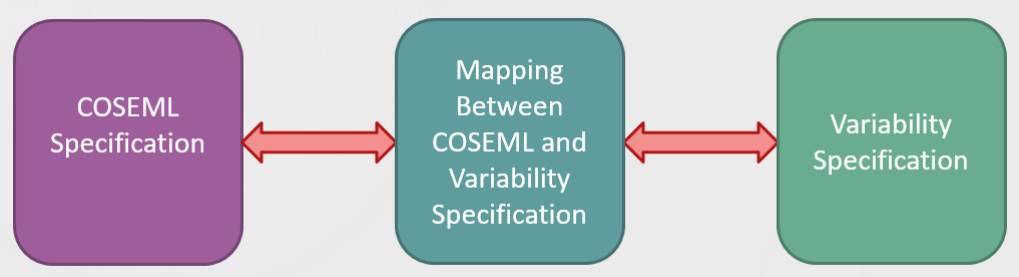

Figure 3.1: Overview of the metamodel.

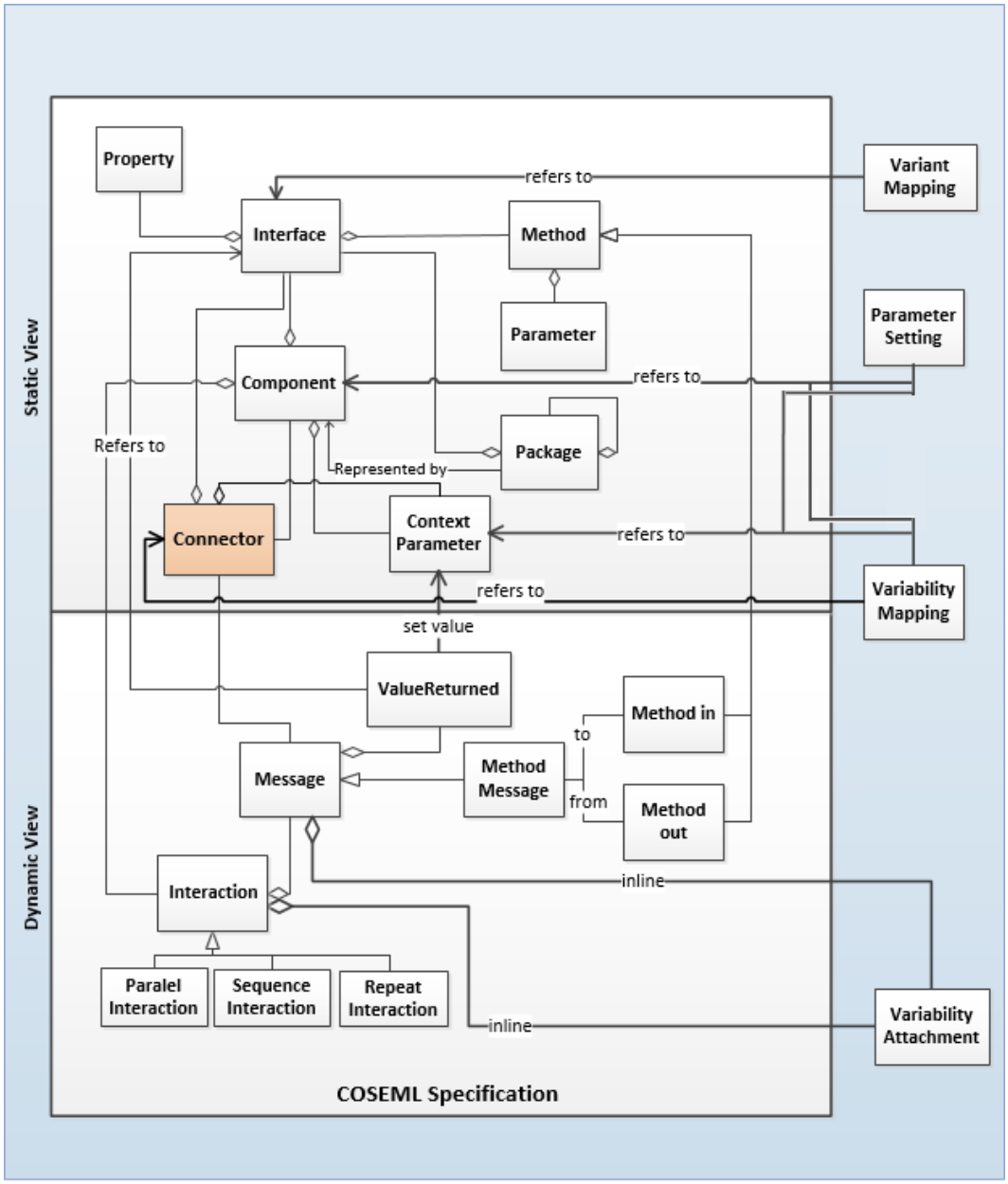

Figure 3.2: The language constructs of the XCOSEML metamodel with connector variability. The boxes at the right-hand side of the figure (outside of the language constructs) are variability mapping constructs. (adapted from [5])

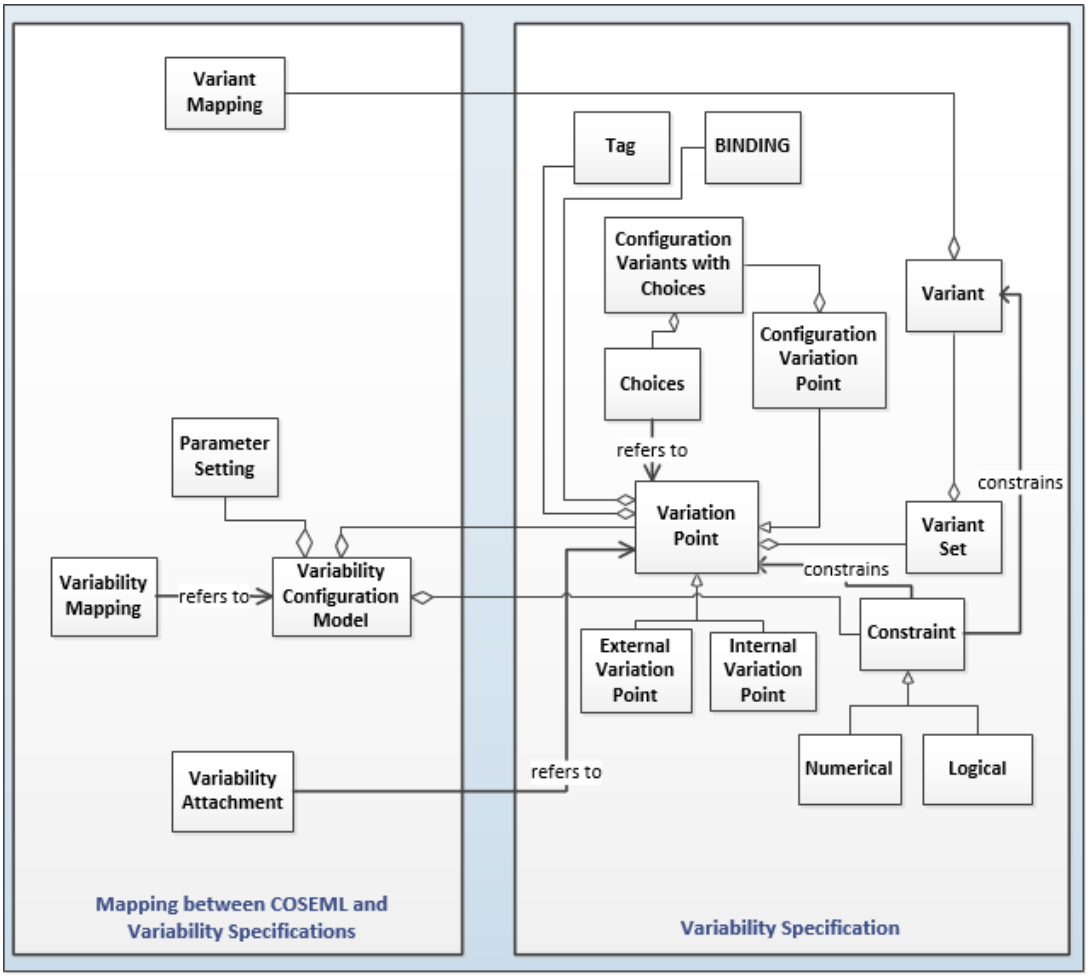

Figure 3.3: Variability Mapping in XCOSEML (adapted from [23]).

In [5], abstract definition of connectors in XCOSEML is extended. The connector definition is enhanced by adding connector message and message operation constructs. A detailed version of the extended XCOSEML grammar is provided in Figure 3.2 and Figure 3.3. Connector message represents messaging between two component interfaces. Message operation represents connectors roles in the communication other than basic communication (if any), such as data transformation. Basic communications such as a procedure call or an event, do not need further operations on data - they simply transfer it.

3.3 Extended Grammar

XCOSEML grammar was first developed in [11]. In this thesis, the grammar is re-developed in ANTLR (Another tool for Language Recognition) [27]. ANTLR plug-ins for different development environments produce parsers for a defined grammar. This eases the process of development of a grammar and making necessary updates. Intellij IDEA is used besides reconstructing the grammar in a new development environment, the following extensions are done for the XCOSEML grammar:

- Connector definition is extended with service type and connector type.
- A connector message is defined to encapsulate communication concerns.
- Connectors additional responsibilities (e.g. data transformation) are represented with the "Operation" keyword.
- Communication protocols for the two interacting components are added to the connector message structure.

In a connector definition, the name of the connector is stated first. Service type and connector type follow this name. Then, one or more connector messages take part in the connector definition.

The name of the connector is a user defined identifier. It can be any string including alphanumeric combinations. XCOSEML does not restrict the syntax of identifiers. However, it is better to follow the naming rules of well-known modeling and programming languages. When naming connectors, it is a good practice to use short forms of components' names that are connected by this connector. Service type and connector type are used based on the categorization defined in the work of Mehta et al. [24]. Using this categorization in the connector definition is an idea of Oussalah et al. [26]. In this thesis, this idea is combined with the advantages of using variability. Service type indicates the connector's purpose of usage. This aim can be achieved through different types of connectors. For example, "procedure call" connector can be used for "communication". However, each type of connector does not have only one service type. In other words, a kind of connector can be used for different purposes. For example, "procedure call" can also be used for "coordination". In

XCOSEML, service type and connector type specifications allow developers to find required connectors easily for their system. Search process becomes easy and it helps development of connectors for specific purposes.

3.4 Case Study: Disaster Management System

A Disaster Management System (DMS) is used as a case study to illustrate the usage of new abilities of XCOSEML. It is a kind of cyber-physical system where humans and various systems work together to manage the emergency situation to lessen the effects of it. This includes understanding the type of the disaster and its effects with all aspects, locating people who need help, organizing various teams to help people and control the situation, and using the time and resources effectively. Figure 3.4 depicts the disaster management system environment.

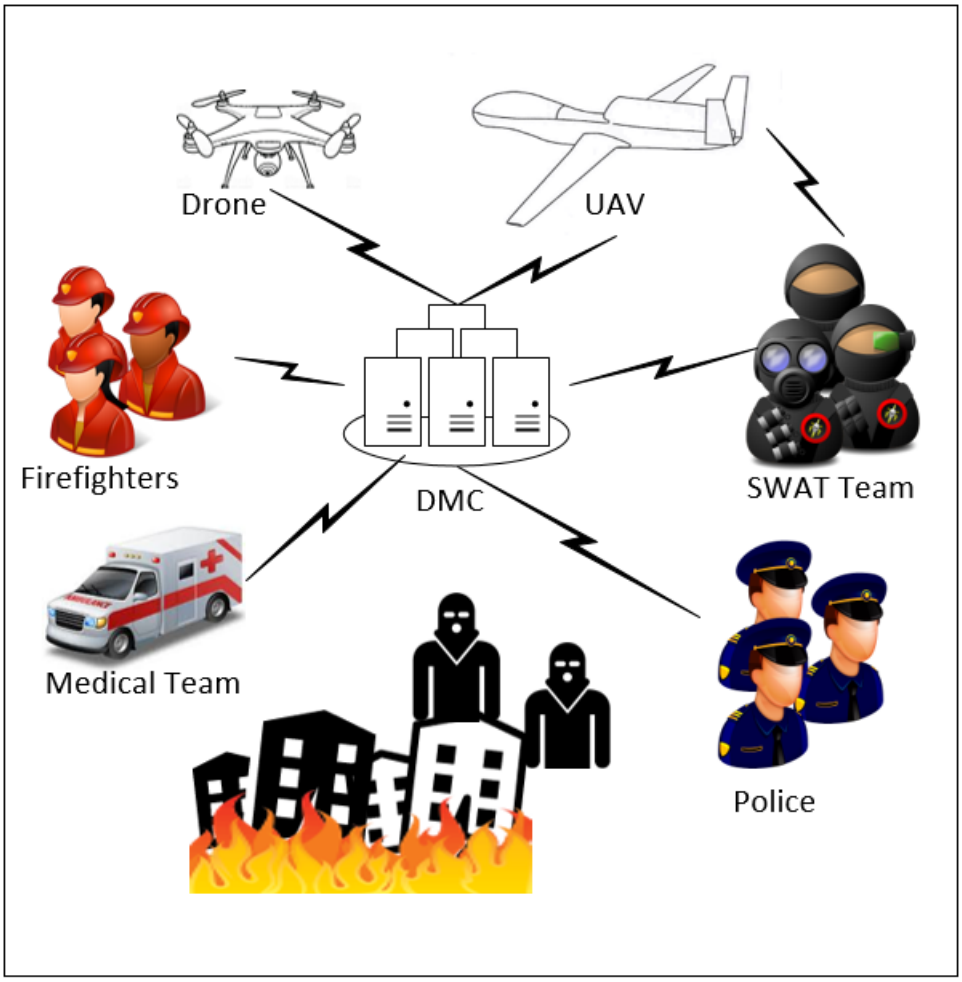

Figure 3.4: Disaster Management System environment.

Disaster Management Center (DMC) coordinates the system. UAVs, sensors and cameras are used to collect data from the environment. When a disaster is detected or reported by the staff, DMC activates necessary teams.

3.5 Modeling Disaster Management System in XCOSEML

Package: XCOSEML representation of the package which contains disaster management system assets named "DMS_pkg" is given in Table 3.1.

Table 3.1: XCOSEML representation of DMS_pkg package.

```
1 Package DMS_pkg
2   includedComponents DMC_comp Drone_comp FireFighter_comp
        MedicalTeam_comp Police_comp SWAT_comp UAV_comp
3   includedConnectors DMC_Drone_conn DMC_FF_conn DMC_MT_conn
        DMC_Police_conn DMC_SWAT_conn DMC_UAV_conn
        SWAT_UAV_conn
4   ConfigurationInterface DMS_conf
5   CompositionSpecification DMS_cmps
```

It can be seen that the package "DMS_pkg" which represents DMS includes all included components and connectors in the system. Configuration interface and composition specification is also included in the package.

Component: The components used to model the system are DMC "(DMC_comp)", Drone "(Drone_comp)", UAV "(UAV_comp)", FireFighter "(FireFighter_comp)", Police "(Police_comp)", SWAT "(SWAT_comp)" and MedicalTeam "(MedicalTeam_comp)". Each of the components has their own interfaces keeping their methods. XCOSEML representation of the DMC component is given in Table 3.2.

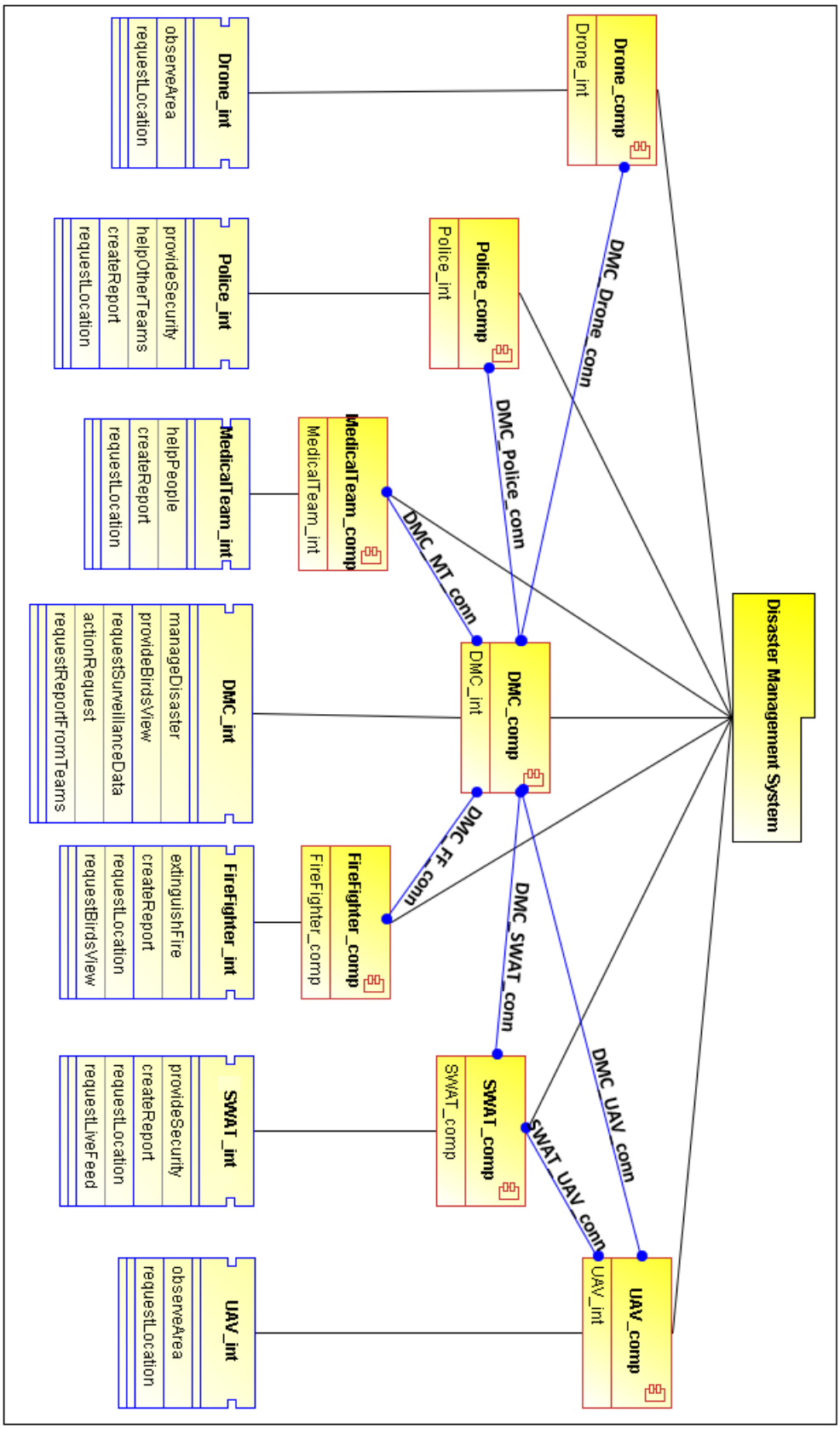

Figure 3.5: COSEML model for the DMS system.

Table 3.2: XCOSEML representation of DMC_comp component.

```
Component DMC_comp
  Interface DMC_int
```

Interface: Interfaces are aimed to store the methods corresponding to their associated components. Interfaces for DMC and FireFighter components are given in XCOSEML representation in Table 3.3 and Table 3.4 respectively.

Table 3.3: XCOSEML representation of DMC_int interface.

```
Interface DMC_int

Provided Methods
  manageDisaster
  provideBirdsView

Required Methods
  requestSurveillanceData
  actionRequest
  requestReportFromTeams
```

Table 3.4: XCOSEML representation of FireFighter_int interface.

```
Interface FireFighter_int

Provided Methods
  extinguishFire
  createReport

Required Methods
  requestLocation
  requestBirdsView
```

Methods in an interface are divided into 2 parts like it was in the COSEML. Interface of the DMC components is given in Table 3.3. The methods shown under the "Provided Methods" tag in lines 4-5 corresponds to "Method In"s of COSEML diagram given in Figure 3.5 with the same names. The methods provided under "Required Methods" tag matches the methods shown in "Method Out" part of the corresponding COSEML diagram of the system.

The methods contained by the "FireFighter" interface are shown in Table 3.4. As it can be seen from the given interfaces, provided methods of a component match required methods of another component such as "provideBirdsView" method shown in line 5 of the Table 3.3 and "requestBirdsView" method shown in line 9 of the Table 3.4.

Connector: Communication among components are done through connectors linking with the in and out method ports of COSEML specifications. These connectors are defined for handling all communication among components; "DMC_Drone_conn", "DMC_FF_conn", "DMC_MT_conn", "DMC_Police_conn", "DMC_SWAT_conn", "DMC_UAV_conn" connectors are created to handle all interactions between the "DMC" and its associated components.

The connector that handles all interactions between DMC and FireFighter components is shown in Table 3.5. Connectors include their service and connector types in their definitions. It is helpful for developers since it provides information and hints regarding with connectors usage and purpose.

Table 3.5: XCOSEML representation of DMC_FF_conn interface.

```
Connector DMC_FF_conn
 ServiceType communication
 ConnectorType procedurecall

 ConnectorMessage extinguishFire_actionRequest {
   RequesterInterface DMC_int
   MethodOut actionRequest
   ResponderInterface FireFighter_int
   MethodIn extinguishFire
   }

 ConnectorMessage createReport_requestReportFromTeams {
   RequesterInterface DMC_int
   MethodOut requestReportFromTeams
   ResponderInterface FireFighter_int
   MethodIn createReport
   }

 ConnectorMessage provideBirdsView_requestBirdsView {
   RequesterInterface FireFighter_int
   MethodOut requestBirdsView
   ResponderInterface DMC_int
   MethodIn provideBirdsView
   }
```

A Connector operates through a structure called “ConnectorMessage”. Body of a connector message refers to interfaces of related components for which it binds a method out from the requester interface to a method in from the responder interface. Name of a connector message is constructed by MethodIn of responder and MethodOut of requester, merged together with an underscore.

The connector which binds DMC and FireFighter components has 3 defined connector messages to handle 3 different operations among these components as it can be seen in Table 3.5. The message “extinguishFire_actionRequest” is used when DMC request firefighters to put off a fire at a certain location. The message “createReport_requestReportFromTeams” is called when DMC requests a situation report from

firefighters. According to variability configuration, “provideBirdsView_requestBirdsView” message can be used to pass bird’s-eye view from DMC to FireFighter.

Configuration: In the configuration file, external-internal variabilities are defined. Configuration file for the DMS is shown in Table 3.6.

Table 3.6: XCOSEML representation of DMS_conf configuration.

```
1  Configuration DMS_conf of Package DMS_pckg
2
3  configurationVP SystemType:
4    external
5    variant standard
6    variant advanced
7  bindingTime devtime
8
9  internalVP SecurityTeamSelection:
10   variant securityViolation
11   variant surveillance
12   bindingTime devtime
```

Between the lines 3-6 of Table 3.6, a couple of scenarios are defined as “standard” and “advanced” to be utilized in the composition. External tag means these selections are open to user and can be selected according to user’s needs while configuring the final product. Between the lines 9-11, internal variants are defined to be mapped to the external variants and manage variation points to be defined in the composition.

Composition: Composition file includes the scenario to be processed. Variability points will be set based on the definitions on the configuration file and the final configured file will be generated by eliminating some variants from the composition file. XCOSEML representation of composition for DMS is given in Table 3.7 and 3.8.

Table 3.7: XCOSEML representation of DMS_cmps composition.

```
Composition DMS_cmps
 import configuration DMS_conf

  has component DMC_comp Drone_comp FireFighter_comp
      MedicalTeam_comp Police_comp SWAT_comp UAV_comp
  has connector DMC_Drone_conn DMC_FF_conn DMC_MT_conn
      DMC_Police_conn DMC_SWAT_conn DMC_UAV_conn
      SWAT_UAV_conn

  mapping
  advanced -> securityViolation
  standard -> surveillance

  Context Parameters
  fireFlag false
  securityViolationFlag false
 Method DMSProcess:
#vp SystemType ifOneSelected(standard)#
    DMC -> Drone (DMC_Drone_conn.
        observeArea_requestSurveillanceData)

#vp SystemType ifSelected(advanced)#
    DMC -> UAV (DMC_UAV_conn.
        observeArea_requestSurveillanceData)
```

Table 3.8: [Continued]XCOSEML representation of DMS_cmps composition..

```
1  guard(fireFlag == true) parallel {
2    sequence {
3      DMC -> FireFighter (DMC_FF_conn.
           extinguishFire_actionRequest)
4      #vp SystemType ifSelected(advanced)#
5        FireFighter -> DMC (DMC_FF_conn.
           provideBirdsView_requestBirdsView)
6      FireFighter -> DMC (DMC_FF_conn.
           createReport_requestReportFromTeams)
7    }
8    sequence {
9      DMC -> MedicalTeam (DMC_MT_conn.
           helpPeople_actionRequest)
10     DMC -> MedicalTeam (DMC_MT_conn.
           createReport_actionRequest)
11   }
12 }
13
14 guard(securityViolationFlag == true) sequnce{
15   #vp SecurityTeamSelection ifSelected(securityViolation)#
16     sequence{
17       DMC -> SWAT (DMC_SWAT_conn.
             provideSecurity_actionRequest)
18       SWAT -> UAV (SWAT_UAV_conn.
             observeArea_requestLiveFeed)
19     }
20   #vp SecurityTeamSelection ifSelected(surveillance)#
21     DMC -> Police (DMC_Police_conn.
           provideSecurity_actionRequest)
22 }
```

In Table 3.7, context parameters to specify fire and security are created as “fireFlag” and “securityViolationFlag”, then they are initialized to “False” in lines between 11-13 of. “#vp” tags are used to define variation points in the system. Variants and their mappings defined in configuration phase are used for determining which variation point is going to be selected in execution. For instance, 2 variation points are defined between lines 15-19. Depending on the selected configuration (standard or advanced), different components and connectors will be used to respond to the events.

If the user selects standard configuration, DMC will alert drone to gather data. If the advanced configuration is selected Unarmed Air Vehicle (UAV) will be used. When a variation point is activated, 2 components communicate through a connector via a connectorMessage defined in the connector body.

The code given between the lines 1-12 of Table 3.8 is used for DMC to respond to the events. In case of a fire, DMC was assigned to send firefighters and medical teams as synchronized. Purpose of the "parallel" tag given in line 21 is to synchronize the communication between DMC-FireFighter and DMC-MedicalTeam. "sequence" tag is tasked with binding multiple messages together. If the fireFlag is toggled to "True", sequences given between the lines 2-12 will be executed. Also, depending on the configuration of this file based on selected scenario, the lines between 4-5 might be deleted from final configured file. That means in case of a fire, DMC will assign firefighters and medical teams to the area. Also, if the advance scenario is selected by the user, DMC will be providing bird's eye view images or video from the UAV. DMC will also request situation reports from both medical teams and firefighters.

If the flag which is used to identify security problems based on the info gathered from drone or UAV is changed to "True", the sequence given between the lines 16-22 will be processed. In the given configuration file advance scenario was mapped to "securityViolation" and standard scenario was mapped to "surveillance". According to the sequence given in this part, police will be assigned to the area in standard scenario. However, if the user selects advanced variation, SWAT teams can be assigned to the area with live steaming support provided by UAV. As it can be seen in line 18 of Table 3.8, the SWAT component has direct access to UAV through the provided "SWAT_UAV_conn" connector.

CHAPTER 4

EXECUTION OF XCOSEML COMPOSITON SPECIFICATION

In this chapter, an effort to execute the Extended Component Based Software Engineering Modeling Language (XCOSEML) composition model is introduced. To achive this goal, the XCOSEML tool is redeveloped and a matching part between configured composition specification and executable code is added. This effort is introduced in section 4.1. Then, implementation of Disaster Management System (DMS) with object-oriented programming is introduced in section 4.2. Finally, execution of "DMS_cmps" composition specification is introduced in section 4.3.

4.1 XCOSEML Tool

The XCOSEML tool has 4 main parts;

- a parser to read XCOSEML files,
- transformation for the model checking tool,
- configuration of variable domain models for customization, and
- matching from XCOSEML models to executable code.

The XCOSEML tool is first introduced in [21] with the parts for parser and configuration. Then, the part for transformation is added in [22] [20]. In this thesis, these three parts are redesigned and recoded. This is done because the desing of the previous version was not modular. Moreover, the previous version of the grammar was developed in Xtext that is a powerful tool working on Eclipse environment. However,

the parser was developed manually and any change of the grammar was required significant effort to update the parser. ANTLR (Another tool for Language Recognition) is prefered for the new version. It otomatically generates the parser. The grammar of the language is designed in a modular fashion which allows generating parsers separately. Therefore, the new tool is easy to update and improve the grammar and the other parts. Grammars for six assets of the XCOSEML are provided in the Appendix A.

As an extension for the tool, a new part for matching XCOSEML models to executable code is introduced in this thesis. This part achives one of the major contribution of the thesis: executable modeling of XCOSEML. Each part of the tool is explained in detail in the below sections.

4.1.1 Parser

Intellij IDEA IDE [19] is used which serves as a good development environment for developing Domain Specific Languages (DSL) with its ANTLR [27] plugin. Figure 4.2 is a screen shot taken from the editor. The structure of the project can be seen from the project window at the left side of the figure. Grammar files of the language are created in ANTLR. To test the language, XCOSEML files are created with the ".xcml" extension. Parser files are created in Java.

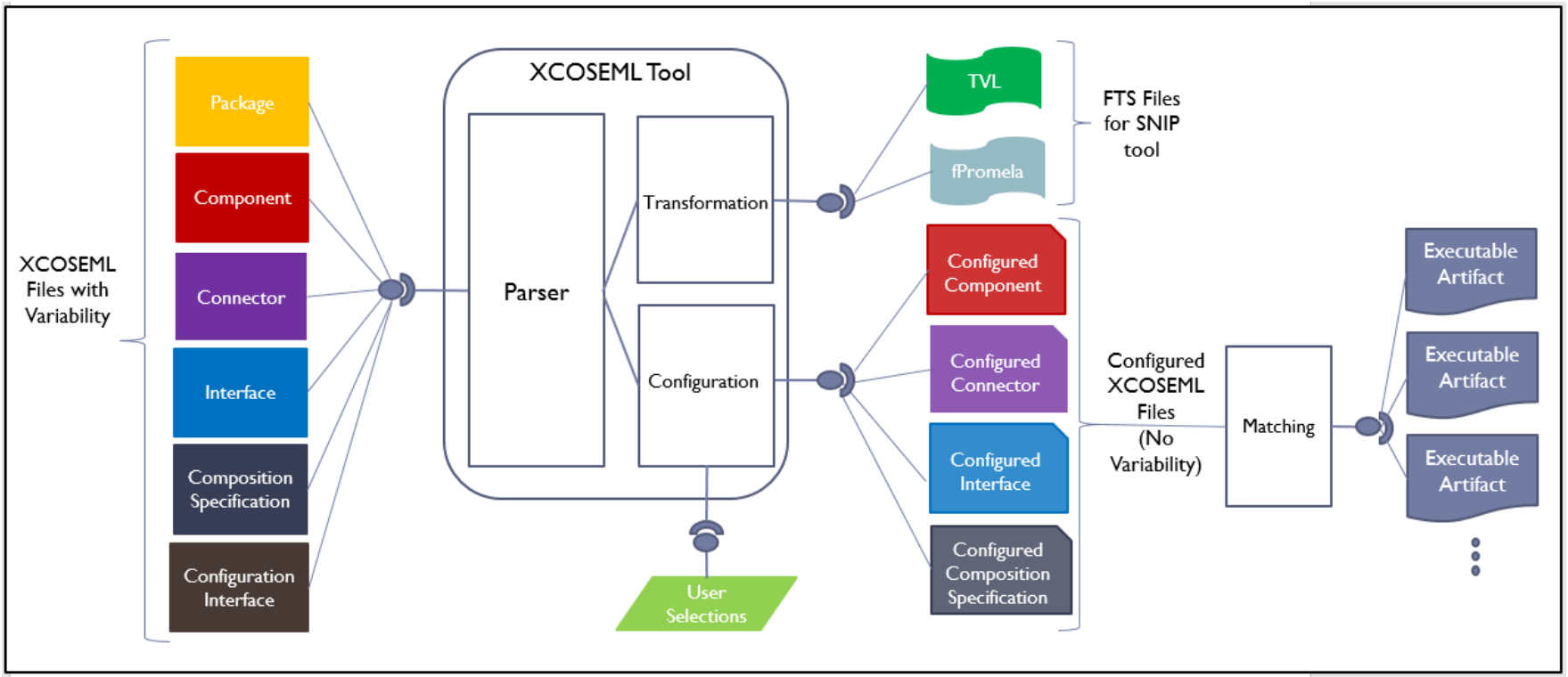

Figure 4.1: XCOSEML Tool.

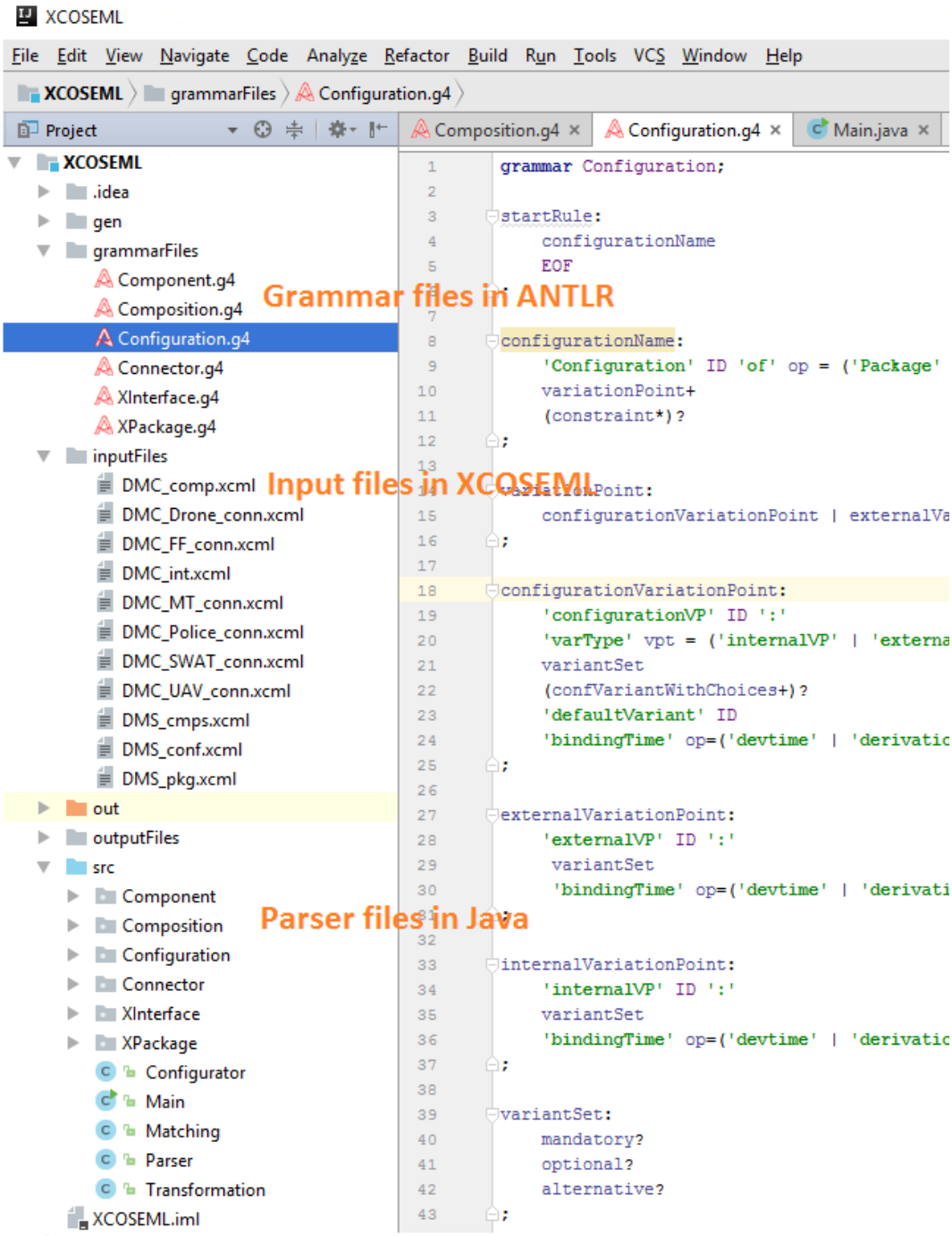

Figure 4.2: XCOSEML tool development environment.

ANTLR is a tool that is designed to deal with textual structures. It contributes to the advancement of programming language design by offering some capabilities for making development of DSL easier. Generating parse trees for the defined context free grammars is one of these capabilities. In this thesis, ANTLR is used to parse

input strings. Each model element has its own structure and they are stored in different files. In ANTLR, we have defined a different grammar for each model element such as connector, interface and component. Then, the parser processes the input files according to the defined grammar for the input type. The final product is gathered via processing the inputs with the generated parse trees.

4.1.2 Transformation

XCOSEML domain models can be verified through model checking with Featured Transition System (FTS) approach [8] before the customization. FTS approach uses SNIP [7] that can handle a domain feature model and a process model separately that is conveniet for XCOSEML. The SNIP tool takes the variability model as a text-based language - Textual Variabiltiy Language (TVL) [6]. The process model of the tool is fPromela, which is an extended version of Promela language of the well-known SPIN model checker [18]. Configuration interface of the XCOSEML is transformed to TVL and composition specification of XCOSEML is transformed to fPromela. Then, domain models are checked against deadlocks and assertions. This transformation process is semi-automated and requires some manual intervention due to some limitations of the source and target languages. Deatailed information for transformation and model checking processes is introduced in [20] [22].

4.1.3 Configuration

In software product lines (SPL) to handle variability we do configuration which refers to the selection of a set of features or parts from the list of all possible elements in the solution space to get a final product. In XCOSEML, variability model resides in a configuration interface. Composition specification has variability tags before an interaction. If the specified variant is chosen, that interaction is included in the final product. These variant selections are done at the specified binding time of the variation point in the configuration interface. Therefore, selected variants are provided by the developer if the binding time is development time. If the binding time were run time, the end user would provide the selected variants. When an interaction is chosen

in the composition specification, components and connectors which take place in that interaction are added to the final product. In the connector message structure, interfaces of the interacting components and their methods are provided. In this way, only desired methods can be included in the final product. This is also the case for choosing a connector message itself. In an interaction different messages of a connector can be used. Therefore, XCOSEML has variability for four assets; component, connector, interface, and composition. Also inner structures of component and connectors are configurable.

4.1.4 Matching

In this thesis, matching part is added to the XCOSEML tool in order to execute the process model of the language. After the configuration of domain process model of the language, i.e. composition specification that contains variability tags, a customized composition file is obtained. This file contains the names of included assets to the system. Matching tool uses these names to map model assets to executables.

4.2 Disaster Management System Implementation in Java

Eclipse [12] is used as an Integrated Development Environment (IDE) with the inclusion of Java 8 version 31. Class diagram of the program is given in 4.3.

One of the challenges with the implementation using connectors was to isolate the components from each other. Which means components would not be able to access other components directly. Since the components are not allowed to access each others methods, we match the method declaration of both requester and responder components in the connector, then allow the connector to be invoked through a connector message as mentioned in section 3.3. When a connector is called with a connector message as a parameter, it determines which component to access according to definitions and methods inside of its body of code. And when it receives the answer from the responder component, it passes the answer to the requester component. The types of these answers may vary according to the type of the connector used based on the defined connector types in [24]. Connectors can also process and change these

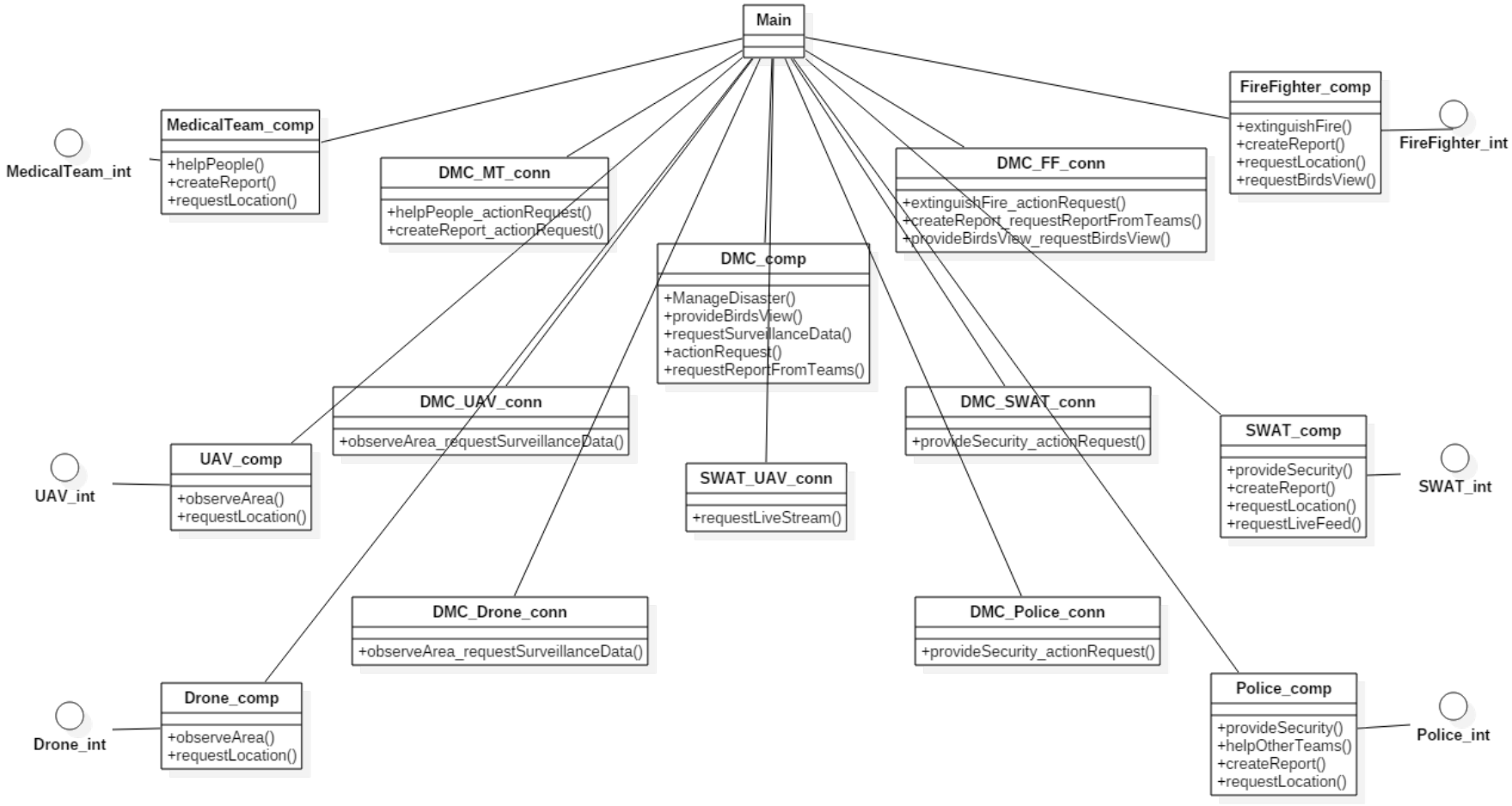

Figure 4.3: Class Diagram of DMS.

answers if they are in their job description. For instance, adapter connectors can be used to compress or enhance the resolution of an image if needed.

4.3 Execution of the DMS_cmps compositon specification

Used components and connectors are specified in the composition file with the "has component" and "has connector" keywords. During the execution of the system, classes will be created for each of these components and connectors according to mentioned keywords. By this way only included components and connectors will be used, and unused ones will be discarded from the final system. Java reflection Aapplication Programming Interface (API) is used to create these classes dynamically in run-time without needing any information of the system by the process. Thus, execution of the composition file is not dependent on context.

Java reflection is used throughout the execution to prevent system to be context or application dependent. Execution starts from the configured composition file which is considered a process model of the system. Our approach for invoking the connector is given in Figure 4.4.

Firstly, process invokes the connector through its "start()" method by sending connector message as a parameter. When the connector is invoked via its "start()" method, it recognizes the connector message. Since the connector includes the requester interface, responder interface, method in and method out information for each connector message, the connector performs the interaction among the components. For this purpose, an object with the type of the connector is defined for every component. Also, an object for each of the component is created with the type of component. This approach is illustrated in Figure 4.5. Then, the connector communicates with the responder component through its required methods by using "method in" defined in the connector message. When the responder component receives the request, a method with the same name with the "method in" is triggered in the provided methods of the responder component. Through this provided method, results will be passed to the connector to be delivered to the requester component.

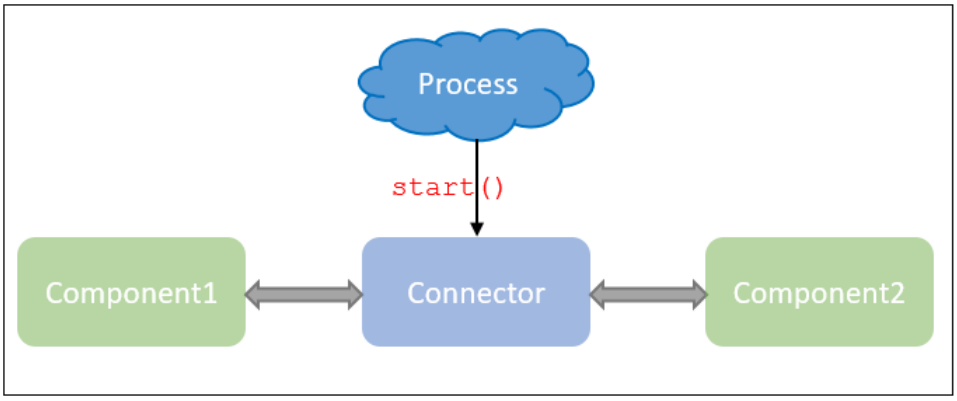

Figure 4.4: Invoking the connectors in proposed approach

Figure 4.5: Illustration of component and connector instances in implementation

The Composition file DMS_comp as shown in Table 3.7 and Table 3.8 is configured with the configuration file DMS_conf as shown in Table 3.6. Generated configuration files for various scenarios such as "Standard", "Advanced" are given in Table 4.1 and Table 4.2 respectively.

A sequence diagram for the configured composition file given in Table 4.1 (Standard scenario) when the global variables "fireFlag" is set to "true" and "securityViolation-Flag" is set to "false" is shown in Figure 4.6. Outputs generated after the execution of the sequence diagram are shown in Table 4.3.

A sequence diagram for the configured composition file given in Table 4.2 (Advanced scenario) when the global variables "fireFlag" is set to "true" and "securityViolation-Flag" is set to "false" is shown in Figure 4.7. Outputs generated after the execution of the sequence diagram are shown in Table 4.4.

A sequence diagram for the configured composition file given in Table 4.1 (Standard scenario) when the global variables "fireFlag" is set to "false" and "securityViolation-Flag" is set to "true" is shown in Figure 4.8. Outputs generated after the execution of the sequence diagram are shown in Table 4.5.

A sequence diagram for the configured composition file given in Table 4.2 (Advanced scenario) when the global variables "fireFlag" is set to "false" and "securityViolation-Flag" is set to "true" is shown in Figure 4.9. Outputs generated after the execution of the sequence diagram are shown in Table 4.6 .

Table 4.1: Configured XCOSEML file for the configuration: "Standard" .

```
1  Composition DMS_cmps
2   import configuration DMS_conf
3
4    has component DMC_comp Drone_comp FireFighter_comp
         MedicalTeam_comp Police_comp
5    has connector DMC_Drone_conn DMC_FF_conn DMC_MT_conn
         DMC_Police_conn
6
7    Context Parameters
8    fireFlag false
9    securityViolationFlag false
10  Method DMSProcess:
11     DMC -> Drone (DMC_Drone_conn.
           observeArea_requestSurveillanceData)
12
13
14 guard(fireFlag == true) parallel {
15   sequence {
16     DMC -> FireFighter (DMC_FF_conn.
           extinguishFire_actionRequest)
17     DMC -> FireFighter (DMC_FF_conn.
           createReport_requestReportFromTeams)
18   }
19   sequence {
20     DMC -> MedicalTeam (DMC_MT_conn.
           helpPeople_actionRequest)
21     DMC -> MedicalTeam (DMC_MT_conn.
           createReport_actionRequest)
22   }
23 }
24
25 guard(securityViolationFlag == true) sequnce{
26     DMC -> Police (DMC_Police_conn.
           provideSecurity_actionRequest)
27 }
```

Table 4.2: Configured XCOSEML file for the configuration: "Advanced" .

```
Composition DMS_cmps
 import configuration DMS_conf

  has component DMC_comp FireFighter_comp MedicalTeam_comp
      SWAT_comp UAV_comp
  has connector DMC_FF_conn DMC_MT_conn DMC_SWAT_conn
      DMC_UAV_conn SWAT_UAV_conn

  Context Parameters
  fireFlag false
  securityViolationFlag false
 Method DMSProcess:
    DMC -> UAV (DMC_UAV_conn.
        observeArea_requestSurveillanceData)

guard(fireFlag == true) parallel {
  sequence {
    DMC -> FireFighter (DMC_FF_conn.
        extinguishFire_actionRequest)
     FireFighter -> DMC (DMC_FF_conn.
        provideBirdsView_requestBirdsView)
    DMC -> FireFighter (DMC_FF_conn.
        createReport_requestReportFromTeams)
  }
  sequence {
    DMC -> MedicalTeam (DMC_MT_conn.
        helpPeople_actionRequest)
    DMC -> MedicalTeam (DMC_MT_conn.
        createReport_actionRequest)
  }
}

guard(securityViolationFlag == true) sequnce{
       DMC -> SWAT (DMC_SWAT_conn.
           provideSecurity_actionRequest)
       SWAT -> UAV (SWAT_UAV_conn.
           observeArea_requestLiveFeed)
     }
```

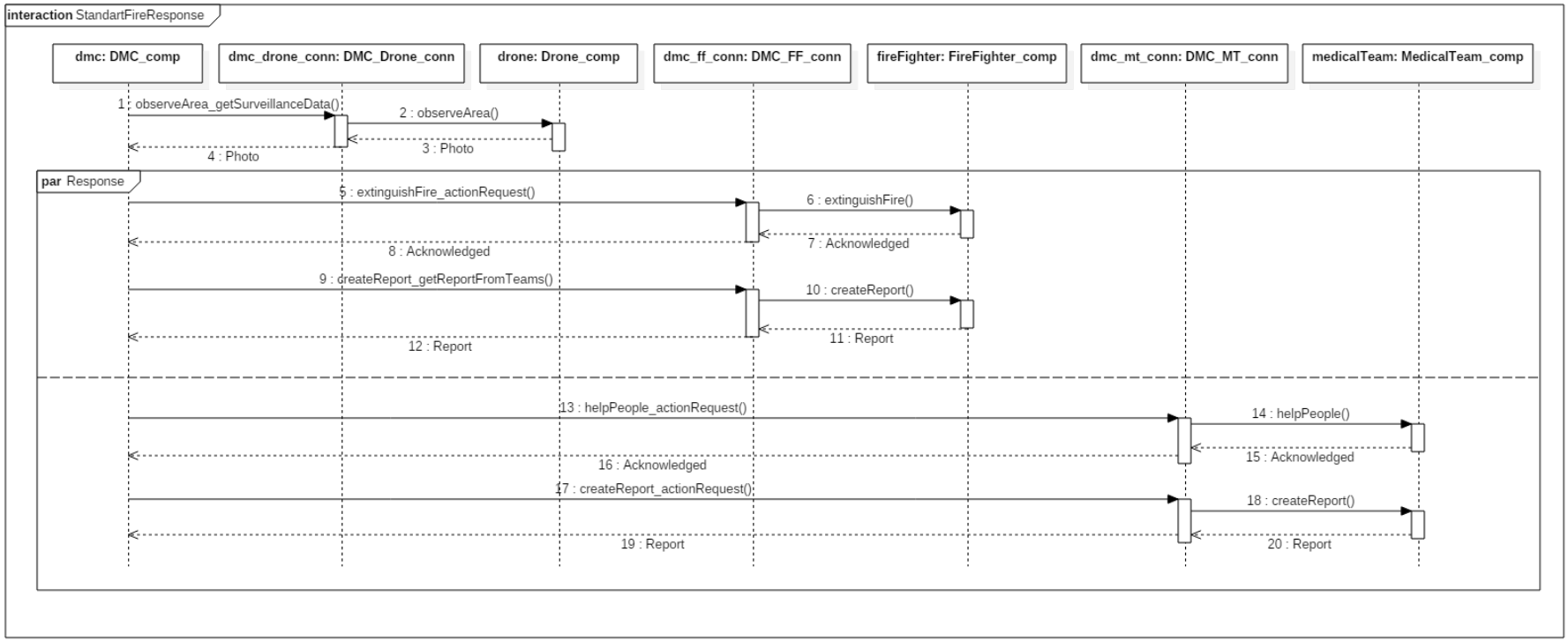

Figure 4.6: Sequence diagram for fire response in standard configuration.

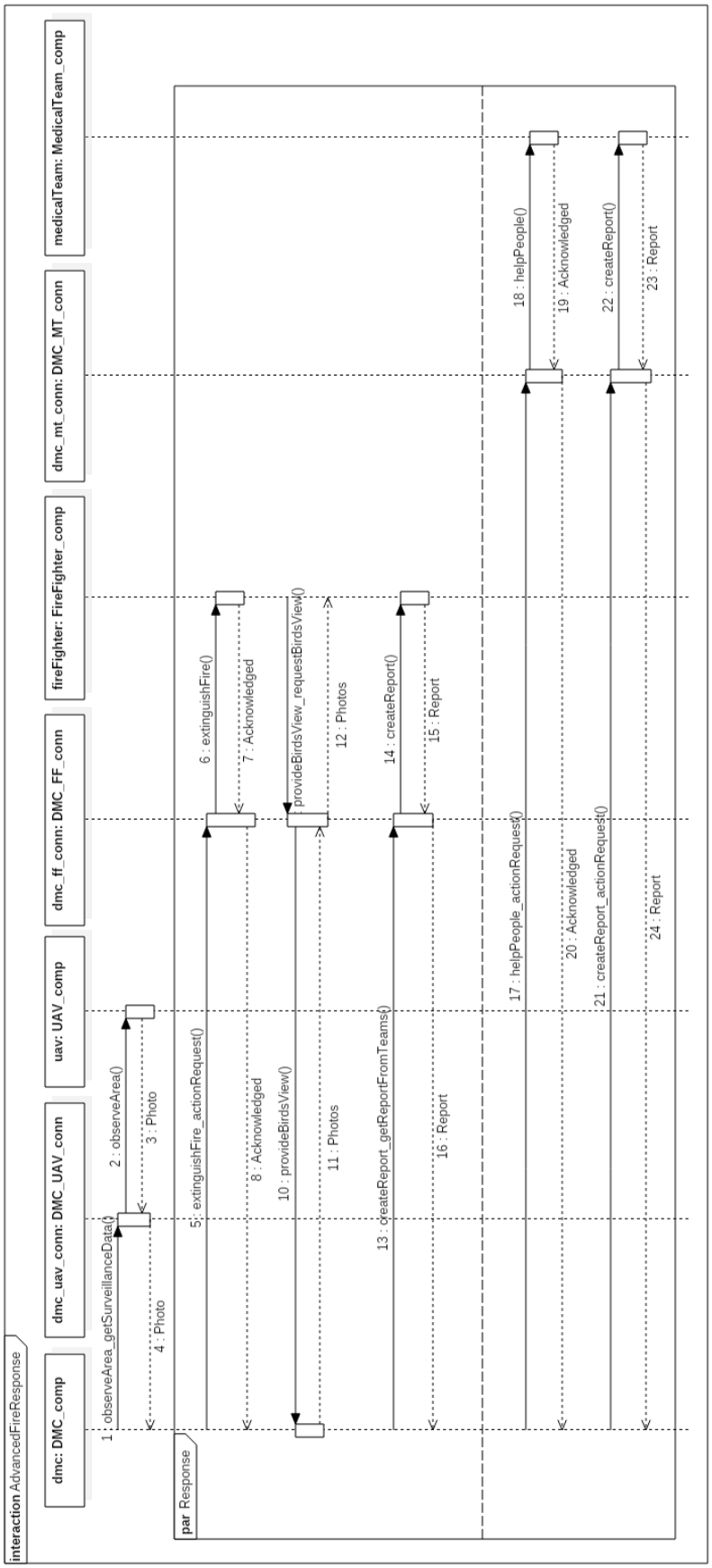

Figure 4.7: Sequence diagram for fire response in advanced configuration.

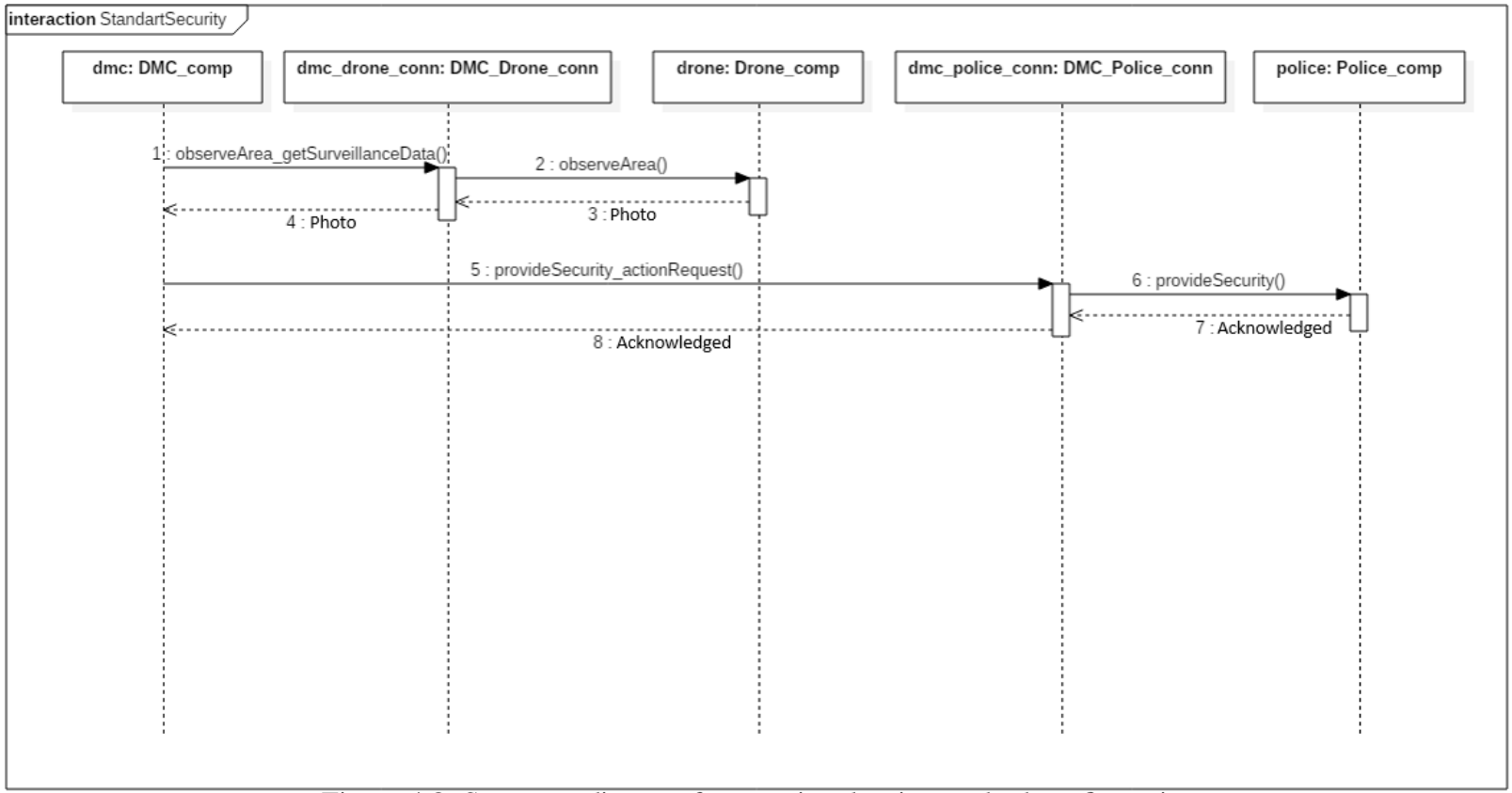

Figure 4.8: Sequence diagram for security alert in standard configuration.

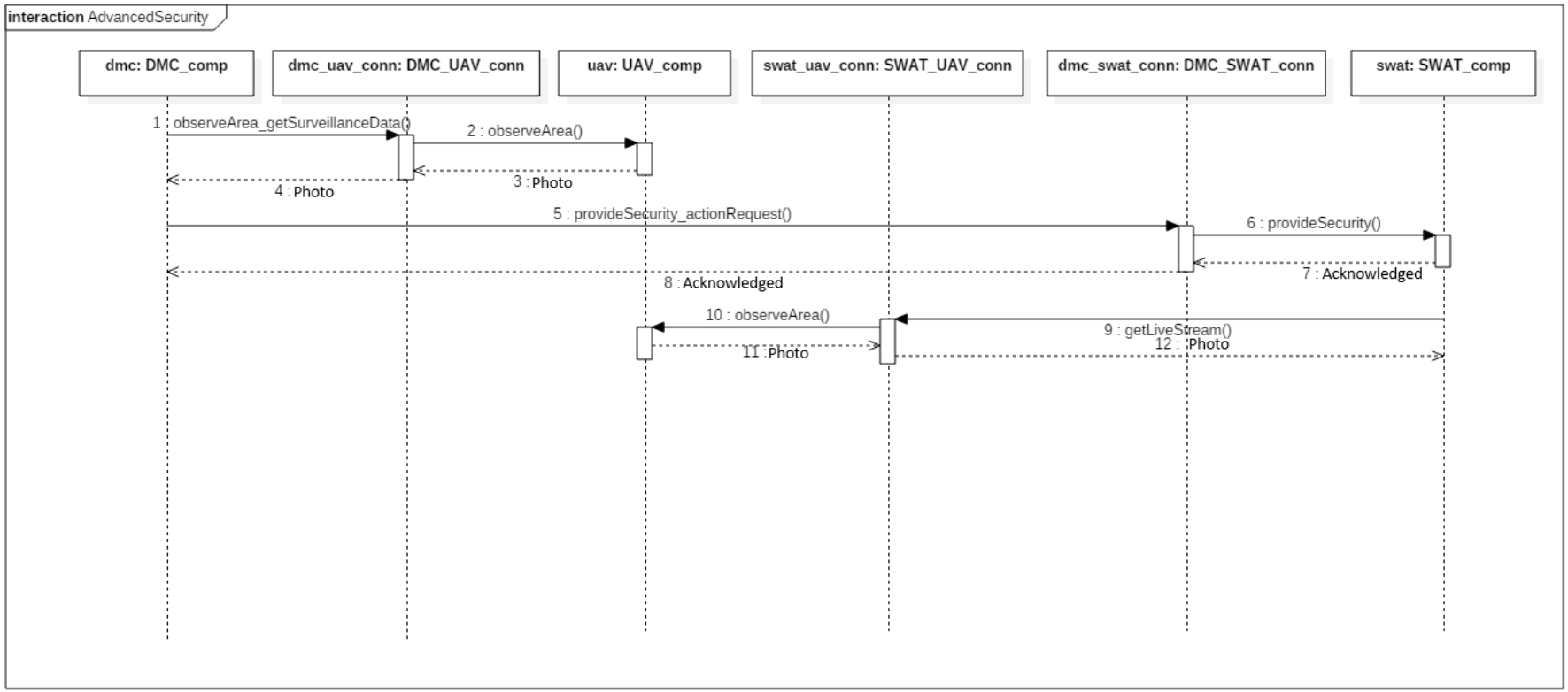

Figure 4.9: Sequence diagram for security alert in advanced configuration.

Table 4.3: Generated outputs after the execution of sequence diagram given in Figure 4.6

```
DMC_Drone_conn: Request for surveillance  received.
drone: Request for surveillance are received.
DMC_Drone_conn: Photos are received.
dmc: Photos are received.
-------------------------------------
DMC_FF_conn: Request for FireFighters received.
ff: Request for FireFighters received.
DMC_FF_conn: FireFighters sent.
dmc: Acknowledged.
-------------------------------------
DMC_FF_conn: Request for fire report received.
ff: Request for fire report received.
DMC_FF_conn: FF Report received.
dmc: FF Report received.
-------------------------------------
DMC_MT_conn: Request for paramedics has received.
mt: Request for paramedics has received.
DMC_MT_conn: Paramedics sent.
dmc: Acknowledged.
-------------------------------------
DMC_MT_conn: Request for medical team report has received.
mt: Request for medical team report has received.
DMC_MT_conn: Report sent.
dmc: Report received.
-------------------------------------
```

Table 4.4: Generated outputs after the execution of sequence diagram given in Figure 4.7

```
DMC_UAV_conn: Request for surveillance  are received.
uav: Request for surveillance  received.
DMC_UAV_conn: Photos are received.
dmc: Photos are received.
-------------------------------------
DMC_FF_conn: Request for FireFighters received.
ff: Request for FireFighters received.
DMC_FF_conn: FireFighters sent.
dmc: Acknowledged.
-------------------------------------
DMC_FF_conn:  Request for BirdsView received.
dmc:  Request for BirdsView received.
DMC_FF_conn: Photos are received.
ff: Photos are received.
-------------------------------------
DMC_FF_conn: Request for fire report received.
ff: Request for fire report received.
DMC_FF_conn: FF Report received.
dmc: FF Report received.
-------------------------------------
DMC_MT_conn: Request for paramedics has received.
mt: Request for paramedics has received.
DMC_MT_conn: Paramedics sent.
dmc: Acknowledged.
-------------------------------------
DMC_MT_conn: Request for medical team report has received.
mt: Request for medical team report has received.
DMC_MT_conn: Report sent.
dmc: Report received.
-------------------------------------
```

Table 4.5: Generated outputs after the execution of sequence diagram given in Figure 4.8

```
DMC_Drone_conn: Request for surveillance  received.
drone: Request for surveillance are received.
DMC_Drone_conn: Photos are received.
dmc: Photos are received.
-------------------------------------
DMC_Police_conn: Security alert has received.
police: Security alert has received.
DMC_Police_conn: Police units are deployed.
dmc: Acknowledged.
-------------------------------------
```

Table 4.6: Generated outputs after the execution of sequence diagram given in Figure 4.9

```
DMC_UAV_conn: Request for surveillance  are received.
uav: Request for surveillance  received.
DMC_UAV_conn: Photos are received.
dmc: Photos are received.
-------------------------------------
DMC_SWAT_conn: Security alert has received.
swat: Security alert has received.
DMC_SWAT_conn: SWAT teams are deployed.
dmc: Acknowledged.
-------------------------------------
SWAT_UAV_conn: Request for live stream received.
uav: Request for live stream received.
SWAT_UAV_conn: Live stream received.
swat: Live stream received.
-------------------------------------
```

CHAPTER 5

CONCLUSION AND FUTURE WORK

5.1 Conclusion

In this thesis, connector variability for component-oriented development is introduced. XCOSEML is extended with variable connectors that declare their service and connector type, and additional operations if exist. A single connector is used for communication between two components. If two components require different types of connectors for different communication concerns, one separate connector must be defined for each concern.

For execution of XCOSEML's process models, a method is proposed by matching model assets to executables. XCOSEML composition specification is parsed, and included components, interfaces, and connectors are found from the set of pre-implemented assets. Then, interactions are triggered through connectors in the order of execution. If a component requires another component to perform its service as a sub-interaction, component itself triggers the connector for this sub-interaction instead of the global process.

The Disaster Management System (DMS) case study proves the applicability of our approach. Pre-built components and connectors of the system can easily be composed to derive a new system by using XCOSEML. Because definitions of components are purified from communication details, their functionality is clearly shown. Moreover, communication details are provided in the connectors explicitly that makes it easy to choose required connector for binding two components. Running the system from XCOSEML's process model - composition specification - is also accomplished. DMS

components and connectors are implemented in Java. Configured composition specification is parsed and included component and connectors are instantiated. Then, connectors are triggered for each interaction.

Dealing with all communication concerns simplifies components which is desirable especially for large scale systems. With a precise definition, reusability for components are increased. Connectors can be reused with a little or no modifications. By undertaking all communication responsibility, connectors collectively constitute a middleware. Therefore, a development methodology by using plug-and-play assets becomes closer.

Besides the mentioned benefits, message trafficking is increased. Traditionally, a component is connected to another component, and this requires request and respond messages. In our approach, connectors are in the middle. They behave like requesters for both of the components. Then they send a message to the other component in the same way. This messaging traffic is not a problem for general information systems. However, for real-time systems this network latency may not be tolerable.

5.2 Future Work

Although the applicability of our approach is proven by the DMS case study, the approach should be tested for large-scale systems. For this purpose, a domain will be chosen and existing components for that domain will be found. Also, additional components will be created. Cyber-pyhsical systems is an appropriate candidate application field for the proposed approach. Diversity of the components and their communication requirements will be handled by connectors. Collaboration of a company would be helpful for industrial scale testing.

Also, a graphical tool is planned to be developed in the future since the use of a graphical tool instead of a textual one would be more user friendly. The design is inspired by Business Process Modelling Language 2.0 (BPMN) and consists of the very similar constructs. The class diagram of the the planned tool is given in Appendix B. Eventually, the aim is to create a software ecosystem where the users can draw a process model and immediately can resolve variation points through the same

tool.

REFERENCES

[1] Robert Allen and David Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 6(3):213–249, 1997.

[2] Felix Bachmann and Paul C Clements. Variability in software product lines. Technical report, DTIC Document, 2005.

[3] Jan Bosch, Gert Florijn, Danny Greefhorst, Juha Kuusela, J Obbink, and Klaus Pohl. Variability issues in software product lines. *Software Product-Family Engineering*, pages 303–338, 2002.

[4] Carlos Cetina, Pau Giner, Joan Fons, and Vicente Pelechano. Autonomic computing through reuse of variability models at runtime: The case of smart homes. *Computer*, 42(10), 2009.

[5] Anil Cetinkaya, M. Çağrı Kaya, and Ali H. Dogru. Enhancing xcoseml with connector variability for component oriented development. In *Proceedings of the 2016 Society for Design and Process Science*, 2016.

[6] Andreas Classen, Quentin Boucher, and Patrick Heymans. A text-based approach to feature modelling: Syntax and semantics of tvl. *Science of Computer Programming*, 76(12):1130–1143, 2011.

[7] 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)*, pages 1–24, 2012.

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

[9] Ali H Dogru. Component oriented software engineering modeling language: Coseml. *Computer Engineering Department, Middle East Technical University, Turkey, TR*, pages 99–3, 1999.

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

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

[12] IDE Eclipse. The eclipse foundation, 2007.

[13] Iris Groher and Rainer Weinreich. Supporting variability management in architecture design and implementation. In *System Sciences (HICSS), 2013 46th Hawaii International Conference on*, pages 4995–5004. IEEE, 2013.

[14] Amina Guendouz, Djamal Bennouar, and Bouira Algeria. Component-based specification of software product line architecture. In *ICAASE*, pages 100–107, 2014.

[15] Arne Haber, Holger Rendel, Bernhard Rumpe, Ina Schaefer, and Frank Van Der Linden. Hierarchical variability modeling for software architectures. In *Software Product Line Conference (SPLC), 2011 15th International*, pages 150–159. IEEE, 2011.

[16] Svein Hallsteinsen, Kurt Geihs, Nearchos Paspallis, Frank Eliassen, Geir Horn, Jorge Lorenzo, Alessandro Mamelli, and George Angelos Papadopoulos. A development framework and methodology for self-adapting applications in ubiquitous computing environments. *Journal of Systems and Software*, 85(12):2840–2859, 2012.

[17] Svein Hallsteinsen, Mike Hinchey, Sooyong Park, and Klaus Schmid. Dynamic software product lines. *Computer*, 41(4), 2008.

[18] Gerard Holzmann. *Spin model checker, the: primer and reference manual*. Addison-Wesley Professional, 2003.

[19] I Jet Brains. Intellij idea. *On-line at www. intellij. com*, 2011.

[20] Muhammed Çağrı Kaya, Mahdi Saeedi Nikoo, Selma Suloglu, and Ali H. Dogru. Towards verification of component compositions incorporating variability. In *Proc SDPS the 20th International Conference on Transformative Science and Engineering, Business and Social Innovation*, 2015.

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

[22] MUHAMMED CAGRI KAYA. Modeling variability in component oriented software engineering. Master's thesis, MIDDLE EAST TECHNICAL UNIVERSITY, 2015.

[23] Muhammed Çağrı Kaya, Alper Karamanlıoğlu, Mahdi Saeedi Nikoo, Sina Entekhabi, Selma Süloğlu, and Ali H. Doğru. Bileşen modellerinde değişkenlik yönetimi yaklaşımlarının incelenmesi. *Ulusal Yazılım Mühendisliği Sempozyumu*, pages 502–513, 2016.

[24] Nikunj R Mehta, Nenad Medvidovic, and Sandeep Phadke. Towards a taxonomy of software connectors. In *Proceedings of the 22nd international conference on Software engineering*, pages 178–187. ACM, 2000.

[25] Peyman Oreizy, Michael M Gorlick, Richard N Taylor, Dennis Heimhigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S Rosenblum, and Alexander L Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems and Their Applications*, 14(3):54–62, 1999.

[26] Mourad Oussalah, Adel Smeda, and Tahar Khammaci. An explicit definition of connectors for component-based software architecture. In *Engineering of Computer-Based Systems, 2004. Proceedings. 11th IEEE International Conference and Workshop on the*, pages 44–51. IEEE, 2004.

[27] Terence Parr. *The definitive ANTLR 4 reference*. Pragmatic Bookshelf, 2013.

[28] Klaus Pohl, Günter Böckle, and Frank J van Der Linden. *Software product line engineering: foundations, principles and techniques*. Springer Science & Business Media, 2005.

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

[30] Jason E Robbins, Nenad Medvidovic, David F Redmiles, and David S Rosenblum. Integrating architecture description languages with a standard design method. In *Proceedings of the 20th international conference on Software engineering*, pages 209–218. IEEE Computer Society, 1998.

[31] Marco Sinnema and Sybren Deelstra. Classifying variability modeling techniques. *Information and Software Technology*, 49(7):717–739, 2007.

[32] Ian Sommerville. *Software engineering*. Pearson, 2016.

[33] Mikael Svahnberg, Jilles Van Gurp, and Jan Bosch. A taxonomy of variability realization techniques. *Software: Practice and experience*, 35(8):705–754, 2005.

[34] Clemens Szyperski, Jan Bosch, and Wolfgang Weck. Component-oriented programming. In *European Conference on Object-Oriented Programming*, pages 184–192. Springer, 1999.

[35] Richard N Taylor, Nenad Medvidovic, and Eric M Dashofy. *Software architecture: foundations, theory, and practice*. Wiley Publishing, 2009.

[36] Jilles Van Gurp, Jan Bosch, and Mikael Svahnberg. On the notion of variability in software product lines. In *Software Architecture, 2001. Proceedings. Working IEEE/IFIP Conference on*, pages 45–54. IEEE, 2001.

APPENDIX A

XCOSEML GRAMMAR LISTINGS

A.1 Package

```
1  grammar XPackage;
2
3  startRule:
4      packageName
5      EOF
6  ;
7
8  packageName:
9      'Package' ID
10     include
11     configurationInterface?
12     compositionSpecification?
13 ;
14
15 //This rule corresponds to included components
16 include:
17     'includes' ID+
18 ;
19
20 //For .conf file
21 configurationInterface:
22     'ConfigurationInterface' ID
23 ;
24
25 //For .cmps file
```

```
compositionSpecification:
    'CompositionSpecification' ID
```

A.2 Component

```
grammar Component;

startRule:
    componentName
    EOF
;

componentName:
    'Component' ID
    xinterface
    configurationInterface?
;

xinterface:
    'Interface' ID+
;

configurationInterface:
    'ConfigurationInt' ID
;
```

A.3 Interface

```
grammar XInterface;

startRule:
    interfaceName
    EOF
;

interfaceName:
    'Interface' ID
```

```
10      providedMethods
11      requiredMethods?
12
13  ;
14
15  providedMethods:
16      'Provided Methods'
17      pmethods+
18  ;
19
20  requiredMethods:
21      'Required Methods'
22      rmethods+
23  ;
24
25  pmethods:
26       ID
27       inputParameters?
28       outputParameter?
29  ;
30
31  rmethods:
32       ID
33       inputParameters?
34       outputParameter?
35  ;
36
37  inputParameters:
38      'input' '(' ID+ ')'
39  ;
40
41  outputParameter:
42      'output' ID
43  ;
```

A.4 Connector

```
grammar Connector;

startRule:
    connectorName
    EOF
;

//definition of connector name
connectorName:
    'Connector' ID
    'ServiceType' ID
    'ConnectorType' ID
    connectorMessage+
;

//Connector message structure
connectorMessage:
    'ConnectorMessage' ID '{'
    'RequiredInterface' ID
    'MethodOut' ID
    'ProvidedInterface' ID
    'MethodIn' ID
    ('RequesterProtocol' protocolReq)?
    ('ResponderProtocol' protocolRes)?
    ('Operation' operation)?
    '}'
;

protocolReq:
    ID
;

protocolRes:
    ID
;

operation:
```

```
38      ID
39  ;
```

A.5 Configuration

```
1   grammar Configuration;
2
3   startRule:
4       configurationName
5       EOF
6   ;
7
8   configurationName:
9       'Configuration' ID 'of' op = ('Package' | 'Component')
            ID
10      variationPoint+
11      (constraint*)?
12  ;
13
14  variationPoint:
15      configurationVariationPoint | externalVariationPoint |
            internalVariationPoint
16  ;
17
18  configurationVariationPoint:
19      'configurationVP' ID ':'
20      'varType' vpt = ('internalVP' | 'externalVP')
21      variantSet
22      (confVariantWithChoices+)?
23      'defaultVariant' ID
24      'bindingTime' op=('devtime' | 'derivation' | '
            compilation' | 'linking' | 'start-up' | 'runtime')
25  ;
26
27  //external variation points
28  externalVariationPoint:
29      'externalVP' ID ':'
```

```
30        variantSet
31        'bindingTime' op=('devtime' | 'derivation' | '
              compilation' | 'linking' | 'start-up' | 'runtime')
32  ;
33
34  //internal variation points
35  internalVariationPoint:
36      'internalVP' ID ':'
37      variantSet
38      'bindingTime' op=('devtime' | 'derivation' | '
            compilation' | 'linking' | 'start-up' | 'runtime')
39  ;
40
41  variantSet:
42    mandatory?
43    optional?
44    alternative?
45  ;
46
47  mandatory:
48      'mandatory' ('variant' ID)+
49  ;
50
51  optional:
52      'optional' ('variant' ID)+
53  ;
54
55  alternative:
56        'alternative' ('variant' ID)+
57        ' min:' INT ' max:' INT
58  ;
59
60  constraint:
61      logicalConstraint | numericalConstraint
62  ;
63
64  logicalConstraint:
```

```
65      'Logical Constraint:'
66          ID (str)? op = ('requires' | 'excludes' | 'implies'
                | 'negates') ID ('selectedVariants(' ID+ ('
                min:' INT)? (' max:' INT)?  ')')?
67  ;
68
69  str:
70   ID
71  ;
72
73  numericalConstraint:
74    'Numerical Constraint:'
75        ID ID 'const' ID ID op = ('>' | '>='| '<'| '<=' |  '
              =='|  '!=' | '=') // variation point-variant
76        (STRING |  'valueOf{' ID+ '}') // variant
77  ;
78
79  confVariantWithChoices:
80    'confvariant' ID 'mapping'
81    choice+
82  ;
83
84  choice:
85    'VPName' ID 'selectedVariants(' ID+  (';  min:' INT)? (',
        max:' INT)? ')'
86  ;
```

A.6 Composition

```
1   grammar Composition;
2
3   startRule:
4       compositionName
5       EOF
6   ;
7
8   compositionName:
```

```
9       'Composition' ID
10      configurationImport?
11      componentImport+
12      ('Context Parameters' contextParameter+)?
13      ('Variability Mapping' variabilityMapping+)?
14      compositionMethods+
15  ;
16
17  configurationImport:
18    'import configuration' ID
19  ;
20
21  componentImport:
22    'has' ID ('with configuration' ID)?
23  ;
24
25  contextParameter:
26    ID (INT | STRING | TRUE | FALSE)
27  ;
28
29  variabilityMapping:
30      'VP' ID 'maps configuration' ID 'VP' ID
31      variantMapping+
32  ;
33
34  variantMapping:
35      'Variant' ID 'maps' 'Variant' ID
36  ;
37
38
39  compositionMethods:
40      'Method' ID ':' interaction+
41  ;
42
43  interaction:
44    (simpleInteraction | compositeInteraction)+
45  ;
```

```
46
47 simpleInteraction:
48   variabilityAttachment?
49   guard?
50   ID op = ('->' | '<-') ID '{' ID '.' ID '}'
51
52 ;
53
54 //interaction options
55 compositeInteraction:
56   repeatInteraction | parallelInteraction |
       sequenceInteraction
57 ;
58
59 //for repeat
60 repeatInteraction:
61   variabilityAttachment?
62     guard?
63   'repeat' intConditionSet '('interaction+ ')'
64 ;
65
66 //for parallel
67 parallelInteraction:
68   variabilityAttachment?
69     guard?
70   'parallel (' interaction+ ')'
71 ;
72
73 //for sequence
74 sequenceInteraction:
75   variabilityAttachment?
76     guard?
77     'sequence (' interaction+ ')'
78 ;
79
80 variabilityAttachment:
81   '#vp ' ID /*VP*/ op = (' ifOneSelected(' | '
```

```
        ifAllSelected(' | ' ifSelected(') ID+ ')' '#'
82  ;
83
84  guard:
85      'guard('
86
87      ')'
88  ;
89
90  //setting conditions
91  intConditionSet:
92    intCondition (('or' | 'and') intCondition )*
93  ;
94
95  //setting conditions
96  intCondition:
97    ID ('>' | '<' | '=' | '<=' | '>=' | '!=') INT
98  ;
```

APPENDIX B

A GRAPHICAL TOOL DESIGN AS A FUTURE EXTENSION

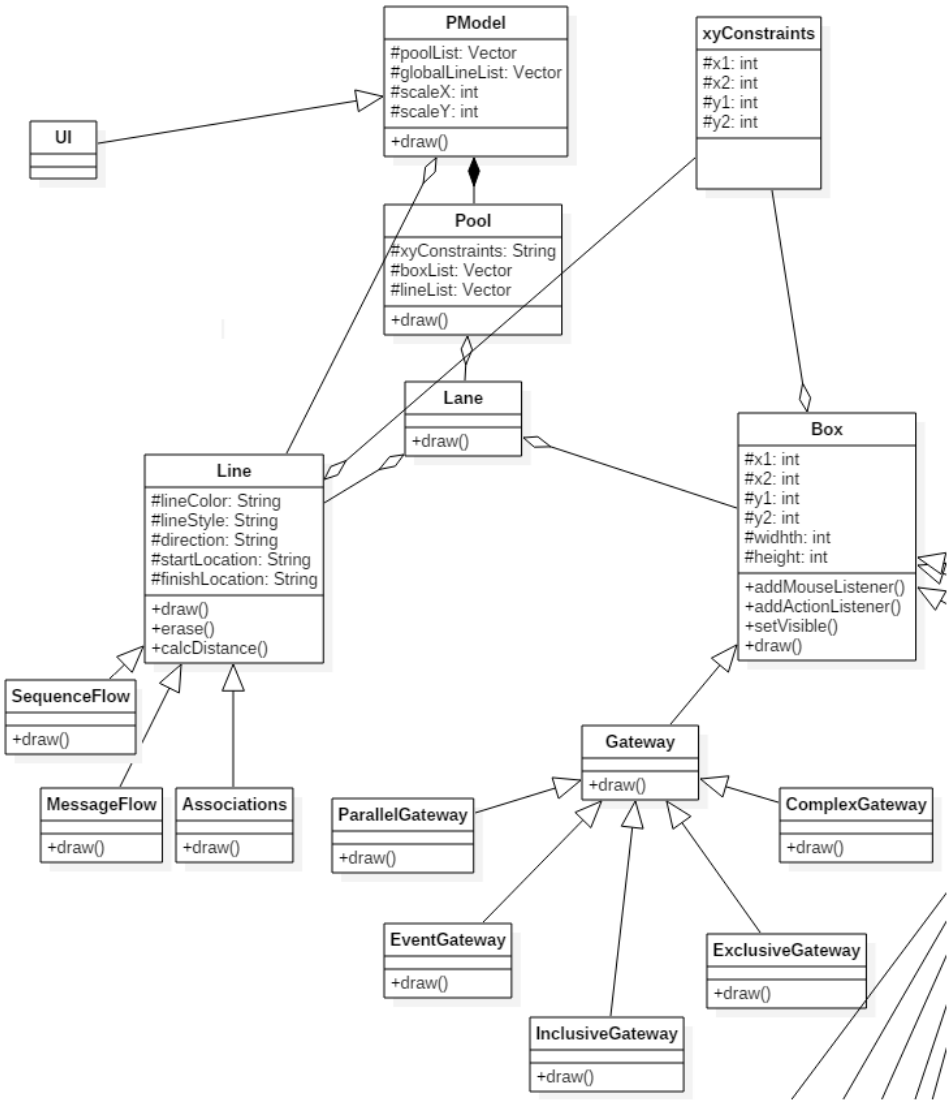

Figure B.1: Class Diagram for the graphical tool designed for future work.

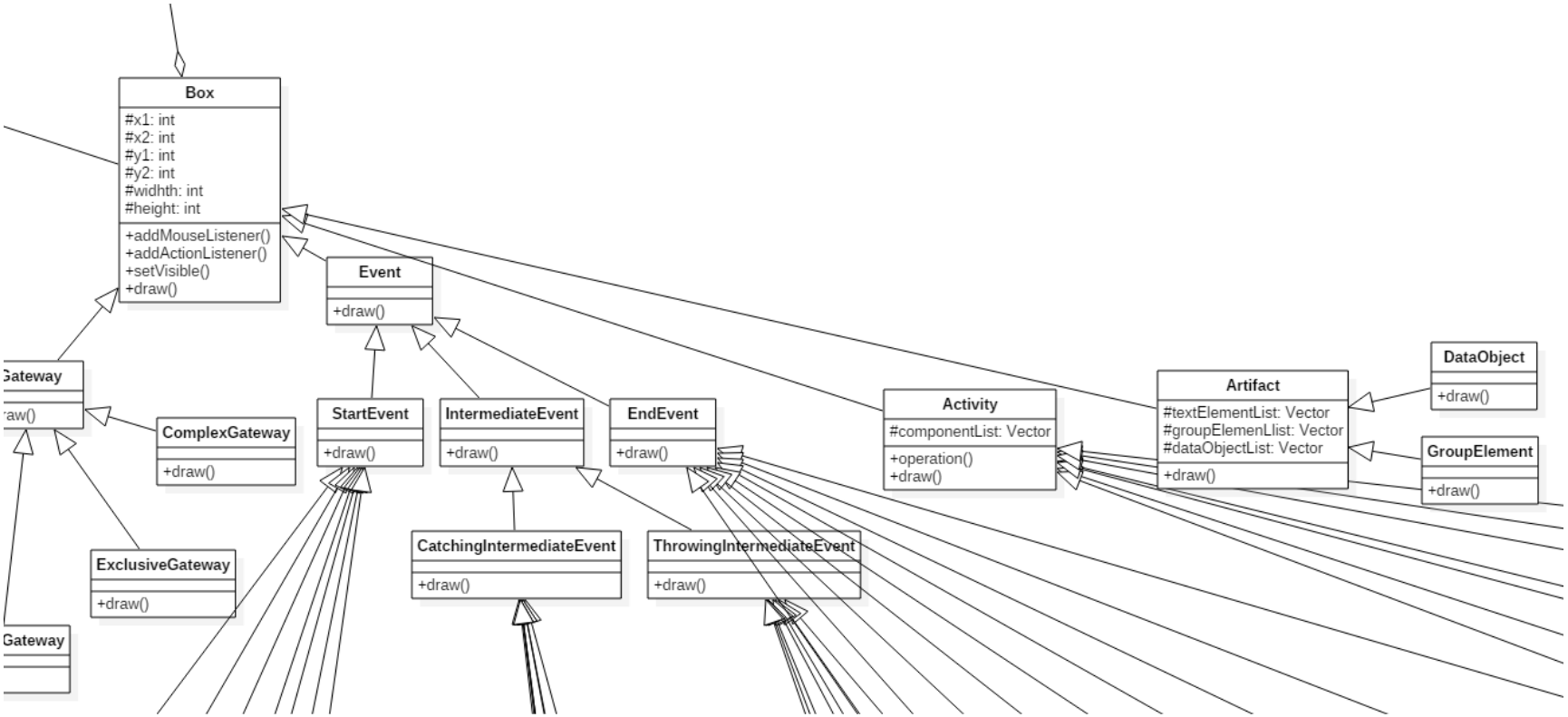

Figure B.2: Class Diagram for the graphical tool designed for future work (continued).