AUTOMATED COMPOSITION OF SEQUENCE DIAGRAMS

by

MOHAMMED IBRAHIM ALWANAIN

A thesis submitted to
The University of Birmingham
for the degree of
DOCTOR OF PHILOSOPHY

School of Computer Science
College of Engineering and Physical Sciences
The University of Birmingham

Abstract

Software design is a significant stage in software development life cycle as it creates a blueprint for the implementation of the software. Design-errors lead to costly and insufficient implementation. Hence, it is crucial to provide solutions to discover the design error in early stage of the system development and solve them. Inspired by various engineering disciplines, the software community proposed the concept of modelling in order to reduce these costly errors. Modelling provides a platform to create an abstract representation of the software systems concluding to the birth of various modelling languages such as Unified Modelling Language (UML), Automata, and Petri Net. Due to the modelling raises the level of abstraction throughout the analysis and design process, it enables the system discovers to efficiently identify errors.

Since modern systems become more complex, models are often produced part-by-part to help reduce the complexity of the design. This often results in partial specifications captured in models focusing on a subset of the system. To produce an overall model of the system, such partial models must be composed together. Model composition is the process of combining partial models to create a single coherent model. Due to manual model composition is error-prone, time-consuming and tedious, it must be replaced by automated model compositions. Given a set of scenarios, it is crucial to check whether these scenarios are consistent and can be combined for a better understanding of the overall behaviour. This thesis presents a novel approach for an automatic composition technique for creating behaviour models, such as a sequence diagram, from partial specifications captured in multiple sequence diagrams with the help of constraint solvers such as Alloy and Z3-SMT.

This thesis addresses the model composition problem by introducing a formal technique for composing behavioural models at the metamodel level through Exact Metamodel Restriction (EMR). In our approach, a sequence diagram can be completely captured by a set of logical constraints at the metamodel level. When composing sequence diagrams, we take the union of the sets of logical constraints for each diagram and additional constraints (composition glue),

which specifies how the models should be glued together to produce the intended composition. At the metamodel level, this gives us the exact instance of the metamodel for the composition. Furthermore, we present a formal semantics for composition using Labelled Event Structures (LES), which guide our model transformation to generate the logical constraints. These, in turn, can be used by constraint solvers to obtain solutions. In addition, we present a comparative study between Alloy and Z3-SMT in the composition of the sequence diagrams from scalability points of view. This study evaluate the performance of both constraint solvers and shows that Alloy is not as scalable as Z3, and for larger sequence diagrams Z3 is a preferred choice.

ACKNOWLEDGEMENTS

First of all, I thank Allah for all the blessings He has given me. I would also like to express my gratitude to all those who helped and supported me in the completion of this thesis. I am particularly grateful to my supervisor, Dr. Behzad Bordbar, for his constant support, motivation, advice and encouragement, which assisted me throughout my Ph.D. research. I would also like to thank Dr. Juliana Bowles for her fruitful collaboration.

In addition, warm thanks go to my office mates: Chris Novakovic, Abdessalam Elhabbash, Ahmed Al-Ajeli, Christopher Hicks, Cory Knapp and Richard Thomas. I am equally grateful to my beloved friends in Birmingham: Dr. Faisal Alrebeish, Dr. Khalid Almeman, Dr. Mohammed Alshammari, Dr. Fahad Alotibi and Mohammed Alharbi.

Neither must I forget to extend my gratitude to the government of Saudi Arabia for funding my studies in the UK. I am especially grateful to Almajmaa University, which has encouraged me and given me the chance to fulfil my ambitions.

Finally, my thanks go out to all members of my family, particularly to my parents for their great support and for all the efforts they have made to keep me eager to achieve my dreams. Moreover, I would like to thank my wife for all the effort she has made to create and maintain the best possible conditions for me to complete my study. I am greatly indebted to her.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

Software engineering is a discipline that provides practical solutions, based on scientific knowledge. It can aid the development of computer software using various methods, languages, tools and procedures. The main goal of software engineering is the cost-effective production of high-quality software systems [137]. The qualities of a software system in this respect include attributes such as efficiency, reliability and maintainability.

Software design is a significant stage in any software developments life cycle, as it interprets the system's requirements and specifications given by various stakeholders into a set of blueprints for the implementation of the software. It is crucial to provide solutions for revealing design errors at an early stage of system development and to resolve them. This is because design errors lead to costly implementation failure, potentially wasting extensive valuable resources, such as time and the cost of fixing the development [83].

Currently, the developments and lifestyle of the modern world increasingly depend on computer software. This pervasiveness has led to the development of more complex systems to handle a wide variety of situations and standard techniques for system development and engineering. However, the expansion of these systems makes the implementation and maintenance of such software increasingly complex. Thus, the process of designing complex software is iterative and it is easy to accidentally overlook design errors. Indeed, the potential for design errors increases with the ever-expanding complexity of software systems. This is due to the limitations of the human mind in managing this complexity [131].

In order to provide a solution for issues surrounding software complexity, the software engineering community has proposed the concept of modelling, which is inspired by mathematical and engineering disciplines. Modelling is the process of generating an abstract representation of a software system, which can be presented in a simple and easily understood format, based on specific modelling languages. A model is normally presented in a graphical or mathematical format, leading to the birth of various modelling languages.

Unified Modelling Language (UML) [116] is one of the commonest modelling languages used to specify various static and dynamic aspects of systems. UML is often referred to as the industry's *'de facto'* language in the modelling of object-oriented systems [141]. It offers rich diagrammatic notations, ideal for supporting the modelling of different views of a system.

UML diagrams can be classified into two main categories: structural and behavioural models. Structural models often focus on particular structural aspects, such as relationships between packages, showing instance specifications or relationships between classes. On the other hand, behavioural models usually emphasise typical scenarios to describe their desired functionality. For example, a class diagram (a structural diagram) is used to model different classes in a system, as well as their attributes and operations and how these classes relate to each another. On the other hand, a sequence diagram (a behavioural diagram) is used to model dynamic interactions, in terms of messages passed between objects in a system. Models in UML are in fact instances of metamodels. A metamodel includes system elements, their relationships and a set of rules, to which every model must conform, in order to be considered as a well-defined model. Metamodels are themselves models, from which models of systems are instantiated [32, 92].

The advantages of software design languages include early assessment of correctness, the requirement for completeness and technical feasibility; all of which help developers avoid the failure of a software project. For example, UML models help developers assess technical feasibility by considering the technical requirements of a proposed project. These technical requirements are then compared to the technical capability of the organisation concerned. UML models also help developers ensure that a system will produce data that is valid, against the values expected (correctness), as well as ensuring that no data is lost in the system design (completeness)

[33].

Despite the advantages of software modelling in UML, however, there are some issues which work against the above-mentioned benefits. Among these is the challenge of model composition, which is the main focus of this thesis.

1.1 Model Composition

As previously established, the process of developing modern systems is gradually becoming more complex. Due to the increase in complexity of such software development processes, multiple models are often used to express various scenarios and viewpoints. This often results in partial specifications captured in models which focus on a subset of a system. However, there are enormous advantages to be gained from system design undertaken with multiple models. One of these is the reduced complexity of designs, whereby designers can focus on specific parts of a system, instead of having to work on a single complex model for an entire system. Furthermore, with the use of multiple models, each model can focus on the needs of a specific stakeholder, in order to gain a better understanding of the software system from that point of view. On the other hand, the advantages of object-oriented design models not only translate easily into object-oriented languages, but also enable system designers to discover errors more easily.

Nevertheless, despite the advantages of separating system design during development, it may be necessary to integrate these models into one, in order to describe the system overview. The process of integrating different models is called *'Model composition'*. Model composition is the process of combining partial models to create a single coherent model, so as to obtain a global representation of a system under construction and to reason over the system as a whole, for the purpose of verification, validation and checking for consistency [34]. Given a set of scenarios, it is crucial to verify that they are consistent and can be combined for a better understanding of overall behaviour.

1.2 Problem Statement

Model composition is a significant step in the development process, which supports software engineers in checking the consistency and understanding overall behaviour. However, UML does not provide support for model composition in its language framework [6]. Instead, various methods of model composition have been introduced in recent years [10, 11, 20, 56, 67, 70, 73, 90, 113, 129, 135, 152, 157]. These methods propose frameworks for composing structural and behavioural models.

For example [56, 67, 129, 135] proposes approaches for composing class diagrams, each of which represents a different way of matching classes, such as by the name of the class [56, 67, 129, 135] or by a signature [56]. On the other hand, [10, 11, 20, 70, 90, 113, 152] present approaches for composing behavioural models, i.e. sequence diagrams [10, 11, 20, 70, 90] and state machine diagrams [113, 152]. In fact, the automated composition of structural models has already been studied [135, 159]. However, the composition of behavioural models is more complex and requires more research to bridge gaps in the automation of their composition [112].

The problems with current techniques can be characterised as follows: 1) Composing systems manually, 2) Only considering the concrete aspect of the models, regardless of the semantic aspects, and 3) Introducing algorithms to produce a composite model from smaller models, originating from partial specifications.

Manual model composition can be done for small models. However, with a large complex model, it is error-prone, time-consuming and tedious [138]. On the other hand, existing approaches treat models as graphical artefacts (concrete aspects), while largely ignoring their semantics and this becomes inadequate for later stages, especially in the process of checking for consistency, which require the model's semantics. Additionally, the existing composition algorithms designed for composing small behavioural models lack the ability to compose complex behaviour, such as parallel or alternative behaviour [7]. The present thesis addresses the above issue in an investigation of the characteristics of behavioural models and through the proposal of an approach to the composition of a behavioural models.

1.3 Proposed Solution

The hypothesis of this research is that constraint solvers, such as SAT and SMT solvers can be used to automatically compose behaviour models. Consequently, this thesis proposes a novel framework for object-oriented modelling composition that composes UML behaviour models automatically, using constraint solvers. Although this approach is applicable to the composition of UML behavioural models, such as Message Sequence Charts (MSC), communication diagrams and sequence diagrams, this thesis focuses specifically on the automated composition of sequence diagrams; one of UML's most popular behavioural models [116]. Sequence diagrams can be used to model complex software systems, as they provide a sequential listing of events and are also able to model parallelism and conflict. Sequence diagrams model system behaviour through interaction or communication between the various objects of a software system.

In this thesis, a sequence diagrams can be completely described by a set of logical constraints on the metamodel. In general, metamodels represent the model elements and their relationships. Logical statements written in the context of metamodels play a key role in expressing the well-definedness of model elements, defining model equality, and so on. As the metamodel represents all compliant models, adding extra logical constraints can restrict the list of models compliant to a metamodel. Furthermore, it is possible to start from a given sequence diagram, M, adding exert logical constraint, $\mathcal{L}$=$\{ \mathcal{L}_1, ..., \mathcal{L}_k \}$, to its metamodel, MM, so that the combination of MM with additional logical constraints, $\mathcal{L}$, can uniquely determine the original sequence diagram, M. We refer to the process of identifying such logical constraints as Exact Metamodel Restriction (EMR).

Logical constraints generated through EMR represent both static (abstract syntax) and semantic (traces of execution) aspects of a sequence diagram. The abstract syntax of a sequence diagram is defined by its metamodel. However, the dynamic interpretation is not given in the metamodel of sequence diagram and must be defined separately. Thus, the dynamic interpretation of the sequence diagram used in EMR employs Labelled Event Structures (LES) [138].

Several possible semantics for sequence diagrams have been defined (see [106] for an

overview). Labelled Event Structures (LESs) are particularly suitable for describing the traces of execution in sequence diagrams, which are able to capture the available notions, such as sequential, parallel and iterative behaviour. For each of the notions, one of the relations available over events is used: causality, conflict and concurrent relationship.

EMR can be used in the automated instantiation of models via constraint solvers. Currently, Alloy [118] and the Z3-SMT solver [42] are widely used for modelling and analysing UML models, because both are supported by automatic tools that are capable of checking a sufficient number of constraints to detect conflict and inconsistency. Alloy is a declarative textual modelling language based on first-order relational logic. It is supported by the Alloy analyser tool, which is an automated constraint solver that transforms Alloy code into Boolean expressions, thus providing analysis through embedded SAT solvers. On the other hand, Z3 is a state-of-the-art SMT solver targeted at solving problems arising in software verification and software analysis. For example, starting from any UML sequence diagram and using a constraint solver, such as the Alloy model finder for the sequence diagram metamodel and a correct set of constraints, Alloy can be used to automatically recreate the original sequence diagram.

Given any two models, M_1 and M_2 representing two partial specifications (e.g. two sequence diagrams) - through EMR, two sets of constraints, $\mathcal{L}_1$ and $\mathcal{L}_2$ are produced on the metamodels which uniquely identify them. To compose these models (M_1, M_2), a *composition glue* is required. This glue consists of a set of syntactically logical constraints, describing how the model elements should be matched. The composition glue matches the *name* and *type* of model elements. If the glue is satisfied and the union of all constraints in the two sets returns *true* (conflict free), the solver will display a solution representing the results of the sequence diagrams composition. Otherwise, it will return $unsat$ and automatically indicate the conflicting statements using SAT Core [140], so that the constraints can be redesigned.

In addition, this approach offers another kind of glue, called *behavioural composition glue*, which provides the designer with a novel way of influencing the composition obtained. This is achieved by specifying behaviour that should never occur, or sequences of events that should occur in a given order. In other words, it allows the designer to prioritise specified behaviour.

The hypothesis for this approach has been evaluated using example scenarios. These examples range from full scale case studies to small scenarios taken from the research literature.

1.4 Thesis Overview

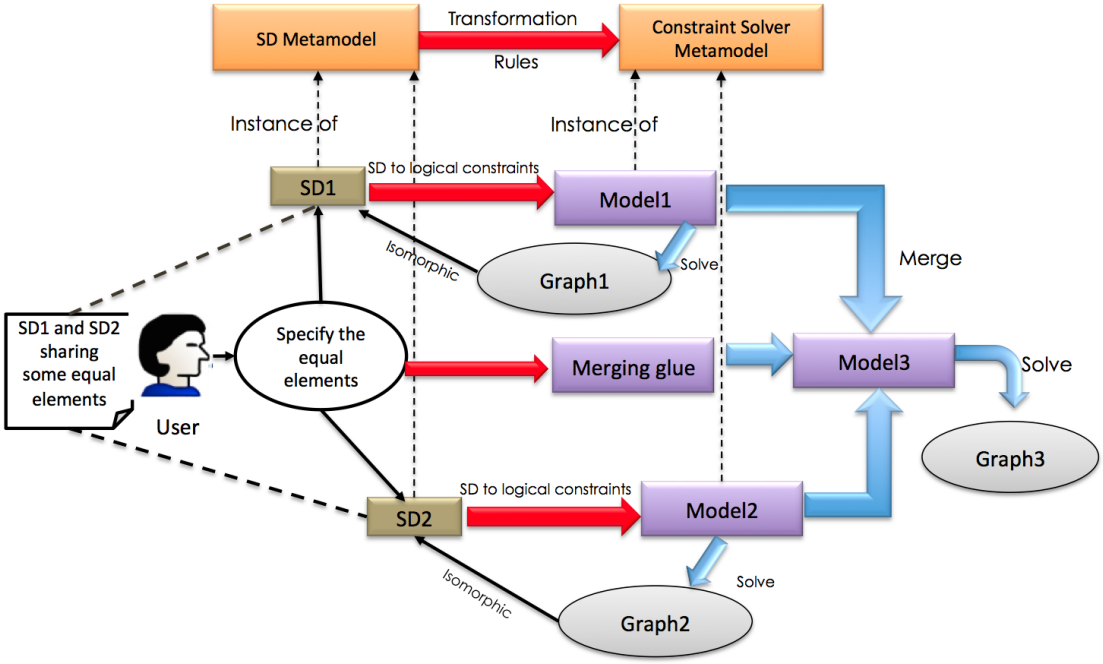


Figure 1.1: Overview of the approach

The main objective of this research is the use of Alloy to automatically compose sequence diagrams. This technique involves three main steps. First, multiple sequence diagrams are automatically transformed into Alloy models. For each sequence diagram, a unique Alloy model is produced. If this is solved, it will have as many solutions as there are possible traces of execution in the original sequence diagram. These traces correspond to those obtained in the underlying semantics of the sequence diagrams used, namely LES.

Second, the Alloy models are composed to produce a single Alloy model. This will contain elements from the individual Alloy models of each sequence diagram, in addition to syntactically logical constraints that specify how the elements are matched and the diagrams should be composed.

In the third step, we use the composed model obtained, that is the conjunction of the overall

logical constraints, to formally check if the sequence diagrams can be composed and obtain the composition of the diagrams automatically, as Figure 1.1 shows. These steps are fully automated in the present *SD2Alloy* tool, implemented using Model Driven Architecture (MDA) techniques [92]. Following composition, a behaviour glue can be added to specify the composed behavioural model. However, during the evaluation of the *SD2Alloy* tool, a performance shortcoming was found in Alloy, when composing more complex sequence diagrams, which can take hours to yield a result. To counteract this weakness, an alternative method of composition using Z3-SMT solver was proposed here, which is a state-of-the-art constraint solver.

In this technique, a number of transformation rules were defined to map the elements of the sequence diagrams and LES metamodels to Z3 metamodels. Using this method, every sequence diagram and its reduced version of the LES model (referred to as *LES'*) are automatically transformed into Z3, which is an instance of a Z3 metamodel. In the LES' model, any events that have not been directly affected by the model behaviour, such as the beginning and end of an CombinedFragment or the initial event of the lifeline to reduce the size of the model, were eliminated. This transformation produced a unique Z3 model, with one solution if solved. This solution was an isomorphic LES' model.

Finally, sets of logical constraints were added, representing the composition glue, matching the common elements of the input models. Similar to Alloy, Z3 was used in this work to formally check if the sequence diagrams can be composed and obtain the composition of the diagrams automatically. Next, a comparative study was conducted between the two methods from the point of view of performance, thus demonstrating that Z3 can resolve the shortcomings of Alloy.

1.5 Contributions of the Thesis

The main contributions of this thesis can be summarised as follows.

- Introducing semantics for sequence diagram composition, using LES.
- A sequence diagram composition framework using Alloy was developed and investigated

focusing on the following points:

1. A subset of the sequence diagram metamodel, expressive enough to model basic components of the sequence diagram, such as lifelines, messages, event occurrence, CombinedFragment and interactionOperands.
2. The transformation rules from the sequence diagram metamodel elements into the Alloy metamodel elements were defined.
3. Two kinds of composition glue were defined: Syntactic glue that matches the elements properties, i.e., name, type and the behaviour glue controlling the behaviour of the composed models.

- The transformation and composition described in this thesis were implemented in a tool called *SD2Alloy*, which facilitates the fully automated transformation and composition of UML sequence diagrams. *SD2Alloy* inherits analytical capabilities of Alloy Analyzer (i.e. it provides support for simulation and the ability to debug the conflict between logical constraints).

- A sequence diagram composition framework using Z3 was developed taking into account the following points:

  1. Transformation rules from the sequence diagrams and LES metamodel elements to Z3 metamodel elements were defined.
  2. Composition glue were defined to compose sequence diagrams.
  3. A case study was used to evaluate and demonstrate the feasibility of the approach presented.

- A comparison study between Alloy Z3 from the point of view of performance was presented.

1.6 Publications

Different aspects of this work have been published during the course of the PhD candidature, resulting in a number of research papers. This thesis should be considered as the definitive reference for the details and ideas presented in the following publications.

- **Conference papers**

  1. Mohammed Alwanain, Behzad Bordbar, and Juliana Bowles. 'Automated composition of sequence diagrams via Alloy'. *2nd International Conference on Model-Driven Engineering and Software Development (MODELSWARD), 2014*. IEEE.
  2. Juliana Bowles, Behzad Bordbar, and Mohammed Alwanain. 'A Logical Approach for Behavioural Composition of Scenario-Based Models'. *The 17th International Conference on Formal Engineering Methods (ICFEM 2015)*. Springer.
  3. Juliana Bowles, Behzad Bordbar, and Mohammed Alwanain. 'Weaving True-Concurrent Aspects using Constraint Solvers'. *16th International Conference on Application of Concurrency to System Design 2016(ACSD)*. IEEE.

- **Book Chapter**

  Juliana Bowles, Mohammed Alwanain, Behzad Bordbar, and Yi Chen. 'Matching and Merging Scenarios Automatically with Alloy'. In *Model-Driven Engineering and Software Development* (pp. 100-116). Springer International Publishing.

1.7 Structure of the Thesis

This thesis is comprised of seven chapters including this introduction.

Chapter 2 begins with an overview of some of the basic concepts related to UML modelling, e.g. the interaction semantics, model composition and technologies used to support composition, especially constraint solvers, such as Alloy and Z3. This is followed by a review that explores current approaches for model composition. The review presents a number of different frameworks used to compose models, as well as the challenges, benefits and trade-offs which must be considered when composing a model. From this background, current approaches using manual composition or algorithms are revealed. Most of these methods involve the introduction of algorithms to produce a composite model from smaller models, originating from partial specifications. The objective of this background is to map out the main activities used to support the composition of dynamic models and identify the gaps in current approaches. It is also revealed by the respective background that the approaches reviewed fail to fully address issues surrounding the automated composition of dynamic models.

In **Chapter 3**, the methodology used for model composition is demonstrated, especially the technique referred to here as Exact Metamodel Restrictions (EMR), which describes mapping between the dynamic models into the logical constraints. This is followed by composition semantics, which lead the composition to produce the expected results. In addition, the syntactic and behaviour glue used for model composition is described.

In **Chapter 4**, sequence diagram composition via Alloy is illustrated. This involves a set of transformation rules that map the sequence diagram elements to Alloy. Logical statements of Alloy are produced through EMR. In addition, this chapter demonstrates the process of composing sequence diagrams via Alloy. This involves the generation of logical statements that represent syntactic and behaviour glue.

In **Chapter 5**, an alternative composition approach using Z3 is presented. The aim of this approach is to resolve the performance issues suffered by Alloy and the use of advantages of Z3 to represent the entire model in one solution. This chapter describes the composition performed

at the level of both the sequence diagram and LES. Moreover, it consists of three main sections. The first demonstrates the mapping between the sequence diagram and LES to Z3; the second demonstrates the composition mechanism and the third evaluates the approach using a case study.

Chapter 6 presents a comparison study between Alloy and Z3, from the perspective of performance.

Chapter 7 then concludes the thesis and points to possible future research avenues.

CHAPTER 2

BACKGROUND MATERIAL AND RELATED WORK

2.1 Overview

This chapter presents an overview of the relevant work and preliminary information for the languages and technologies used throughout this thesis, including Unified Modelling Language (UML), Labelled Event Structure (LES), Alloy, Z3-SMT, and Model Driven Architecture (MDA).

2.2 Models and Metamodels

The primary aim of this thesis is model composition. However, it is first necessary to define the terms, 'system model' and 'metamodel'. The Object Management Group (OMG) defines the term, 'model' as follows:

> "A model is a representation of a part of the function, structure and/or behavior of a system. A model is a specification that is said to be formal when it is based on a language that has a well-defined form ("syntax"), meaning ("semantics"), and possibly rules of analysis, inference, or proof for its constructs. The syntax may be graphical or textual. The semantics might be defined, more or less formally, in terms of things observed in the world being described (e.g., message sends and replies, object states and state changes, etc.), or by translating higher-level language constructs into other constructs that have a well-defined meaning." [92]

The construction of models requires a modelling language capable of defining both structure and semantics. These aspects are defined by a representation known as a metamodel, i.e. a model at a higher level of abstraction that defines the modelling language of a specific mode [32]. The relationship between a model and its metamodel is as follows: a model only contains concepts from the metamodel and satisfies the constraints of that metamodel, while a metamodel can be understood as a collection of models. Furthermore, a metamodel is generally a structural model presented as a UML class diagram; frequently with additional constraints being given in OCL, i.e. UML's constraint language. A metamodel includes system elements, their relationships and a set of rules to which every model must conform in order to be considered well-defined. Every element in the model is an instance of a metamodel element and every element in the metamodel categorises the model elements (Figure 2.1). For example, let us suppose a modelling language, $\mathcal{L}$ has a metamodel, $M_{\mathcal{L}}$. $M_{\mathcal{L}}$ is therefore a model that describes the constructs of the language, $\mathcal{L}$ and every model written in $\mathcal{L}$ must be instance of the metamodel $M_{\mathcal{L}}$.

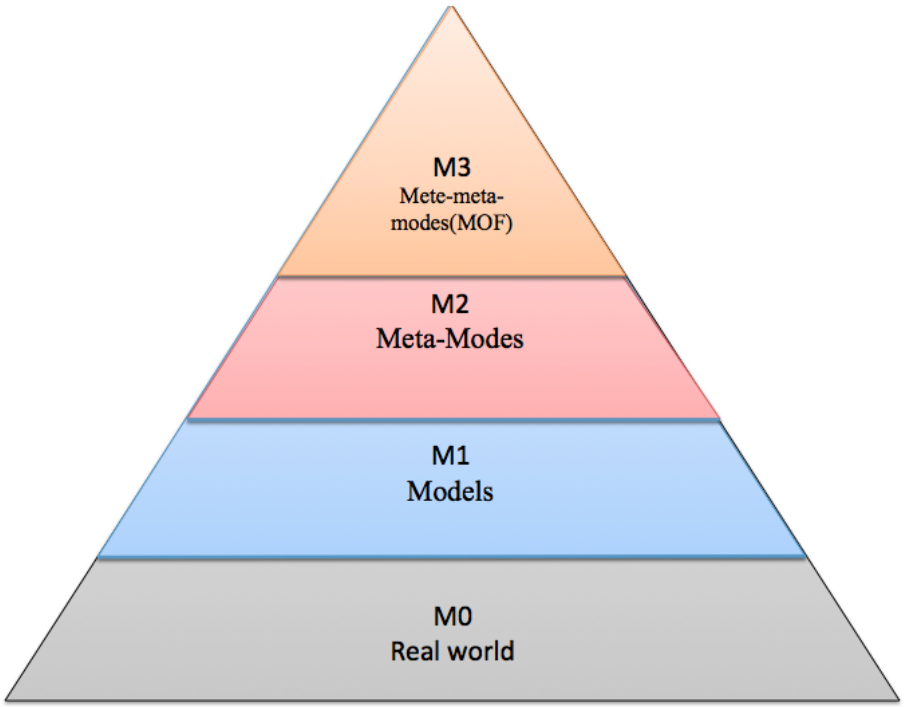


Figure 2.1: The OMG four-layer hierarchy [65]

However, since a metamodel is yet another model, it also has its own metamodel, which

is moreover required to conform. This is called the meta-meta-model (MOF) [115]. MOFs are reflexive (i.e. they define their own elements, structure and semantics), in order to avoid multiplying the levels of abstraction. To clarify the relationship between the model, metamodel and meta-meta-model, Figure 2.2 demonstrates an example adopted from [75]. In Figure 2.2, at model level (M1), the model Person conforms to a metamodel defined at metamodel level (M2). This means (the conformance here) that the model defined in the lower level is an instance of the model defined in the level above. Consequently, the Person model defined at model level (M1) is an instance of its metamodel defined at level (M2). For example, 'Person' is an instance of $Class$, and 'age' is an instance of $Attribute$.

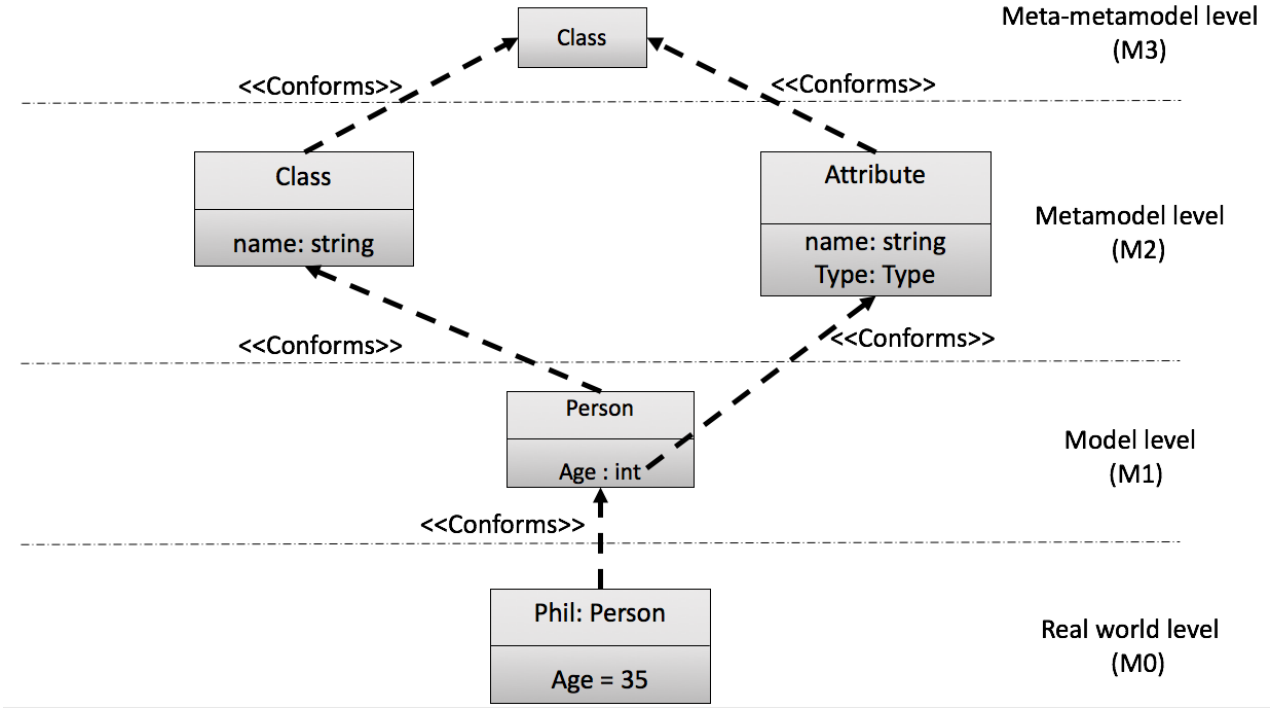


Figure 2.2: An example of model and its metamodel [75]

In the following section, one of the commonest modelling languages is described, namely UML.

2.3 Unified Modelling Language

Unified Modelling Language (UML) [116] is a modelling language defined by the Object Management Group (OMG) [117] and is widely accepted as the *"de facto"* standard for software modelling. UML offers various diagrammatic notations to aid in the modelling of different views of a system. The majority of recently created large systems have been designed using UML. According to Pender [128], approximately 70 % of object-oriented software projects have been designed using UML. Moreover, Miles et al. [107] state six major advantages of using UML:

1. It is a well-defined language: All elements used in UML have a strongly defined meaning explained in [116], which offers clear guidance to illustrate how UML can be used to model different parts of systems.

2. Concise: The notations used in the language are simple and easy.

3. Comprehensive: UML describes different aspects of a system: static (structure) and dynamic (behaviour). This is due to UML being built as a collection of languages.

4. Scalable: UML is implemented to be strong enough to model large complex systems. It is also flexible to model smaller-scaled systems.

5. Built on lesson-learned: UML is developed on the best practices in previous systems' modelling methods. It is also constantly improving.

6. UML is developed by open standards with active contributions from vendors and academics all over the world. These standards ensure that UML promotes interoperability and discourages a vendor monopoly.

UML consists of two groups of diagrams: structural diagrams and behavioural diagrams. Structural diagrams focus on the architectural construct of the system, such as the display of relationships between classes or instance specifications. On the other hand, behavioural diagrams usually emphasise typical scenarios to describe the desired functionality of the system [94].

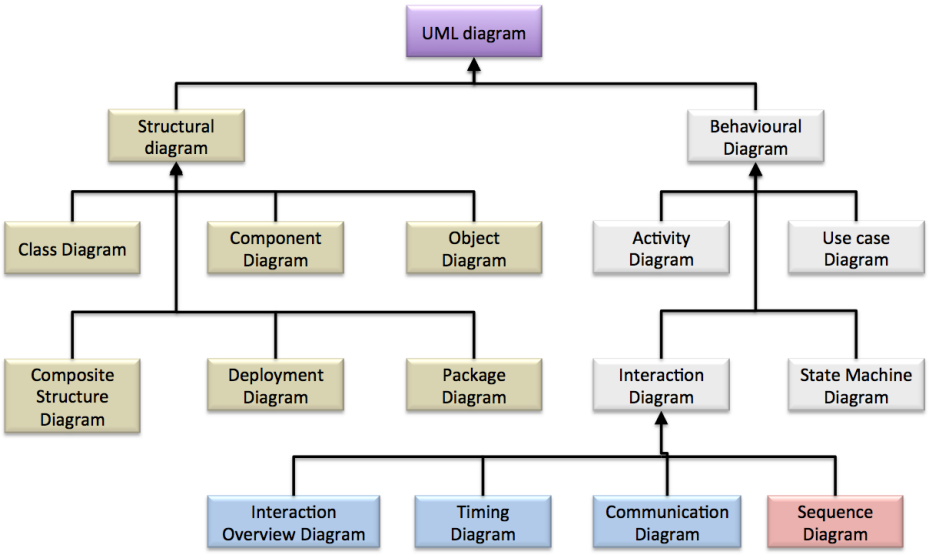


Figure 2.3: The classification of UML diagrams

Figure 2.3 classifies six structural diagrams and seven behavioural diagrams, four of which are further classified as interaction diagrams. Table 2.1 lists and describes the different types of UML diagrams.

Table 2.1: UML diagrams

UML diagram	Description
Class Diagram (page 21 of [116])	A class diagram is one of the UML static diagrams that depict the structure of the system. The representation of the system involves classes and the relationships between them. Classes in this diagram are represented as a box shape with three fields. The upper field contains the class name, while the middle field holds class attributes. Finally, the bottom field consists of the methods that are associated with the class.

Object Diagram (page 21 of [116])	An object diagram is a graph of instance, which consists of objects and data values. It is an instance of a class diagram, which is used to represent a complete or partial view of a system modelled at a specific time. Object diagrams are used to show examples of the data's structure.
Component Diagram (page 145 of [116])	A component diagram aims to depict the relation between system components. These diagrams are used to illustrate the structure of arbitrarily complex systems.
Composite Structure Diagram (page 167 of [116])	One of the static diagrams is a Composite Structure Diagram. It shows both the collaboration between the classes, including the internal structure of classes. This diagram might include a description of the $parts$ (roles) of various instances, the $ports$, that is, points of connections between the classes and the connectors that are used to bind the entities together.
Deployment Diagram (page 199 of [116])	A deployment diagram is also a static diagram used for modelling the physical deployment of $artefacts$. In UML, artefacts can be a model file, a source file, a table, or even a Word document, among others. For instance, to describe a particular website, a deployment diagram can illustrate the necessary hardware and software components and also how the different pieces are connected.

Package Diagram (page 21 of [116])	Finally, a package diagram illustrates the dependencies between packages in a system model. It is normally used to show the architecture of a system using layers and communication between them.
Use Case Diagram (page 597 of [116])	A use case diagram is one of the behavioural diagrams that represent the overview of functionality in a system. A use case diagram consists of $actors$ and the dependencies between use cases. A use case diagram is often applied to capture the requirements of a system.
Activity Diagram (page 303 of [116])	An activity diagram is aimed at representing the workflow of system activities. It models a different kind of behaviour, such as choice and parallel behaviour.
Sequence Diagram (page 473 of [116])	A sequence diagram is a type of interaction diagram used to depict the communication between various object instances in a system. Sequence diagrams are capable of modelling different kinds of behaviour, such as a sequence of events in a system, parallelism, loops and alternatives (choice).
State Machine Diagram (page 535 of [116])	A state machine diagram is an extension of state charts [76]. There are two kinds of state machines: behavioural and protocol. Behavioural state machines are used to specify the behaviour of various model elements, while a protocol state machine is used to express usage protocols.

Communication Diagram (page 473 of [116])	A communication diagram is a type of interaction diagram that is simplified from the collaboration diagram found in previous versions of UML. It is commonly regarded as a combination of class diagrams, sequence diagrams and 'use case' diagrams, as it is capable of modelling both static structures and dynamic behaviours.
Interaction Overview Diagram (page 473 of [116])	An interaction overview diagram is similar to an activity diagram in terms of modelling the control flow in a system using types of interaction diagram (communication diagrams, interaction overview diagrams, sequence diagrams and timing diagrams).
Timing Diagram (page 473 of [116])	A timing diagram is a type of behavioural diagram that focuses on timing properties. The horizontal axis of the timing diagram represents time, increasing from left to right, whereas the vertical axis represents the object instances.

Table 2.1 describes the various types of UML diagrams. However, with reference to the highlighted element in Figure 2.3, the next section provides a more detailed view of a specific diagram type that will be used throughout this thesis: sequence diagrams.

2.3.1 Sequence Diagrams

The sequence diagram is an interaction diagram adopted from the Message Sequence Chart (MSC) [106]. Sequence diagrams are described in UMLs superstructure specification [116] using both a concrete and abstract syntax. The concrete syntax consists of the graphical notation for a sequence diagram, whereas the abstract syntax is given by a metamodel, which defines all the elements of a sequence diagram model and their possible relationships. An *instance* of the

metamodel corresponds to a concrete sequence diagram.

2.3.1.1 Concrete Syntax

An interaction captured by a sequence diagram involves a group of objects, which exchange messages between each other to achieve a particular goal. Each object has a vertical dashed line called a lifeline showing the existence of the object at a particular time.

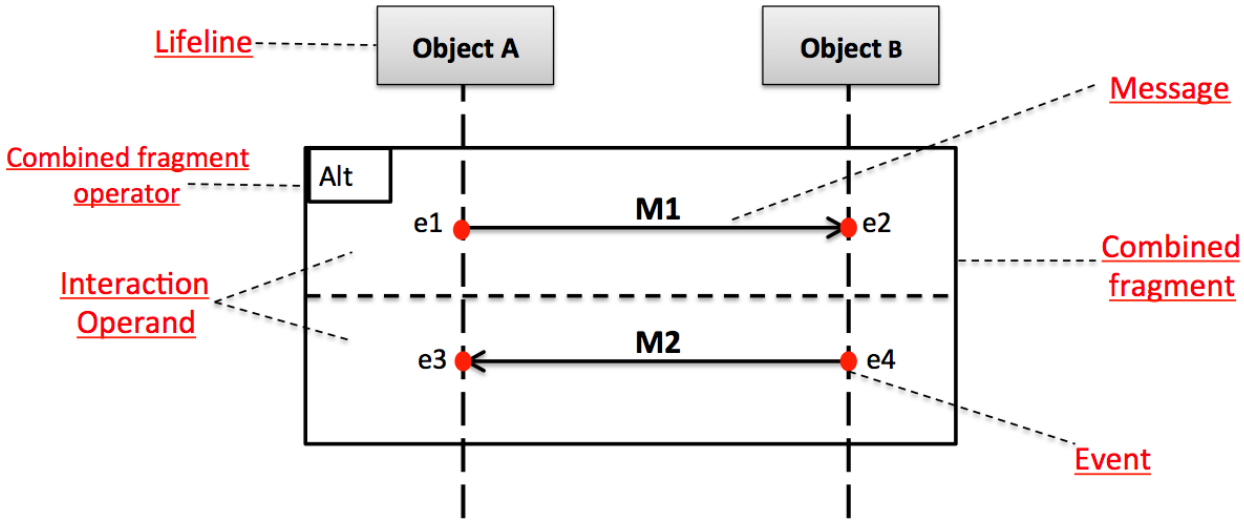


Figure 2.4: Example of a sequence diagram

A message is a communication between two objects shown as an arrow connecting the respective lifelines: that is, the underlying send and receive events of the message. An interaction between several objects consists of one or more messages but may be given further structure through so-called CombinedFragment (Figure 2.4). There are several kinds of CombinedFragments including seq (sequential behaviour), alt (alternative behaviour), par (parallel behaviour), neg (forbidden behaviour), $assert$ (mandatory behaviour), $loop$ (iterative behaviour), and so on [116]. Depending on the operator used, a CombinedFragment consists of one or more InteractionOperands. In the case of the alt CombinedFragment, each InteractionOperand describes a choice of behaviour. Only one of the alternative InteractionOperands is executed if the guard expression (if present) evaluates it as 'true'. If more than one InteractionOperand has a guard that is evaluated as true, one of the InteractionOperands is selected nondeterministically for execution. In the case of the par CombinedFragment, there is a parallel

merge between the behaviours of the InteractionOperands. The event occurrences of the different InteractionOperands can be interleaved in any way as long as the ordering imposed by each InteractionOperand as such is preserved. Finally, interaction fragments can be nested producing expressive and complex scenarios of execution. Consider the following sequence diagrams, which show a slightly adapted example from [70]. Figure. 2.5 (left) reveals an interaction with two consecutive CombinedFragments (a parallel followed by an alternative CombinedFragment), while Figure 2.5 (right) shows a different interaction involving the same instances and a few additional messages. In both diagrams, all messages are sent asynchronously between objects a and b (only new messages are sent by b to a).

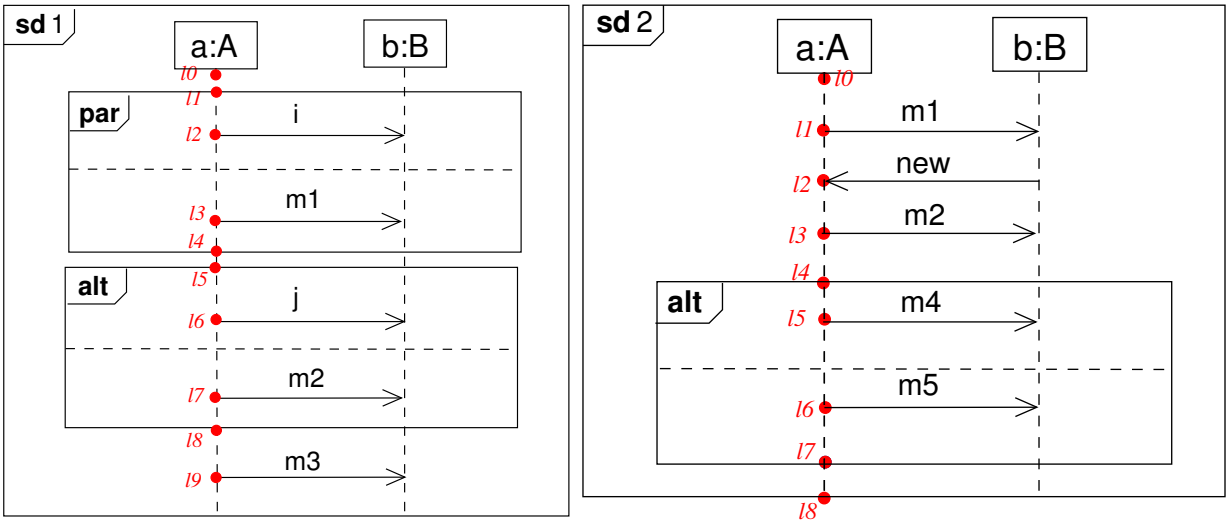


Figure 2.5: Two sequence diagrams with fragments involving the same object instances

Points along the lifeline are called locations (terminology borrowed from LSCs [77]) and denote the occurrence of events. The order of locations along a lifeline is significant, denoting, in general, the order in which the corresponding events occur. The importance of locations is described in section 2.3.2.1. In particular, the distinction between the syntactic notions of a location on a sequence diagram from its semantic counterpart of an event will be clarified. In Figure 2.5 (left), messages $m1$ and i are sent/received in parallel followed by message j or message $m2$ (alternative) and further followed by message $m3$ (irrespective of the previous alternative chosen). In Figure 2.5 (right), three messages are sent/received before reaching an alternative CombinedFragment and choosing between messages $m4$ or $m5$. These diagrams

will be used throughout this thesis as a running example to demonstrate the transformation and composition of sequence diagrams automatically via constraint solvers.

2.3.1.2 Abstract Syntax

A metamodel can be understood as a model of a collection of models. A metamodel is usually a structural model given as a UML class diagram often with additional constraints given in UML's constraint language OCL. Metamodels can be built for both static and dynamic models but focus only on the structural aspects of the model. The metamodel of a sequence diagram, also known as an *interaction*, shows the structure of such a diagram in terms of the model elements present and their relationships. A dynamic interpretation is not given in the metamodel and instead must be defined separately, using the semantic methods available, such as LES [138], Labelled Transition System (LTS) [15, 147], Petri Nets [110] and so on. The UML superstructure specification [116] defines the interaction's metamodel a package which shows different elements and their relationships separately, using multiple diagrams. To make the presentation simpler, a subset of the metamodel for interactions has been used (Figure 2.6). The main notions required for the present thesis have hereby been captured.

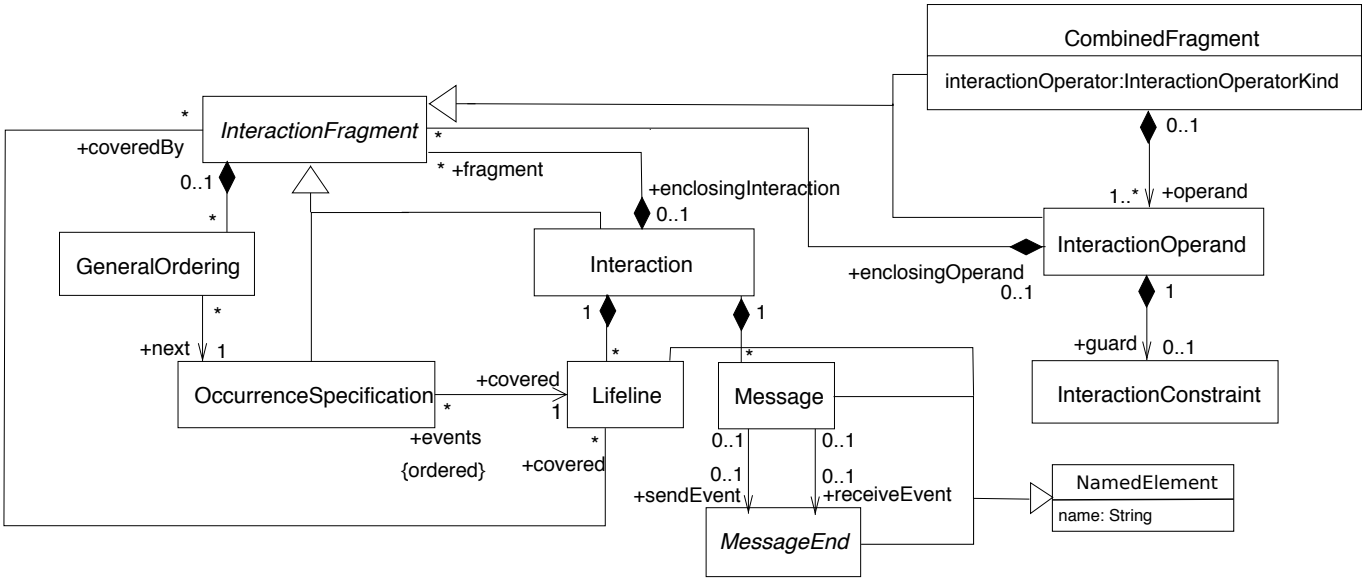


Figure 2.6: The Interactions Metamodel[116]

An `Interaction` contains zero or more `Lifelines`, `Messages` and `InteractionFrag`

ments. A Message usually has a sendEvent MessageEnd and a receiveEvent MessageEnd associated with it. In this thesis, we assume that a MessageEnd (an abstract class) is always a special kind of OccurrenceSpecification called MessageOccurrenceSpecification (not shown). It is possible for a Message to have been found (or similarly, lost), in which case it does not have a sendEvent or a receiveEvent . Moreover, a lost message can be described as a message where the sendEvent is known, but there is no receiveEvent. It is interpreted as if the message never reached its destination. A found Message is a message where the receiveEvent is known, but there is no sendEvent. It is interpreted as if the origin of the message is outside the scope of the description.

Lifeline has attributes for the name and class associated with the object that is denoted by the lifeline. An InteractionFragment is an abstract class, which is further specialised into an OccurrenceSpecification, an Interaction, aCombinedFragment or an InteractionOperand. The locations mentioned in the LES section correspond to OccurrenceSpecifications. These are the ordered events that cover a Lifeline. A GeneralOrdering represents a binary relation between two OccurrenceSpecifications. The metamodel contains relations before and after, but in this thesis, both relations are modelled as a relation next. A CombinedFragment has an attribute InteractionOperator of enumeration type InteractionOperatorKind (par, alt, seq, loop, assert, and so on), and contains one or more interactionOperands. An InteractionOperand may have a guard, which is an InteractionConstraint. An InteractionOp erand encloses either a set of events (OccurrenceSpecifications), an Interaction or another Combined Fragment, indicating nesting of fragments. An instance of the metamodel represents a concrete interaction or sequence diagram. The interaction from Figure 2.5-sd1 can be captured using the abstract syntax as an instance of the metamodel, as partially depicted in Figure 2.7-sd1.

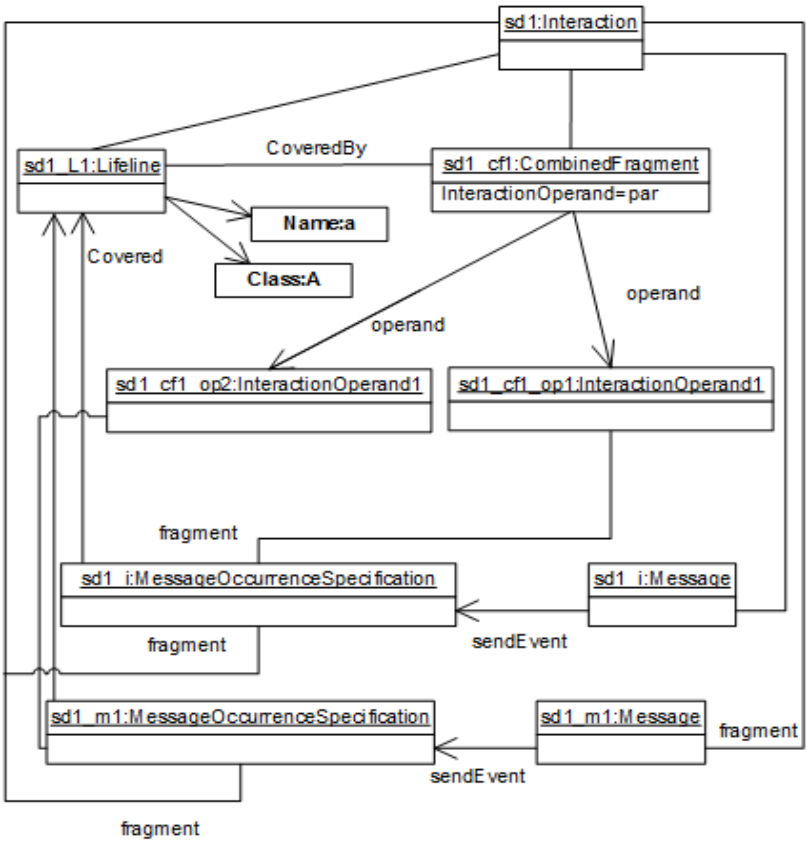


Figure 2.7: Extract of the abstract syntax of sd1

A full abstract syntax of sd1 (Figure 2.5-$sd1$) is very large. Therefore, only part of it is shown in Figure 2.7, which illustrates how the abstract syntax of the sequence diagram is modelled. This model shows that the `Interaction` is a container for all other elements. The messages `sd1_i`, `sd1_m1` are linked to their send `MessageOccurrenceSpecifications`. Both `MessageOccurrenceSpecifications` are covered by lifeline `L1`. The lifeline `L1` are connected to the CombinedFragment called `sd1_cf1` with `InteractionOperator=par` whereas the CombinedFragment contains two interactionOperands `sd1_cf1_op1` and `sd1_cf1_op2` with no guards were considered. Both interactionOperands contained one message; more specifically, `sd1_cf1_op1` contains message `sd1_i`, and `sd1_cf1_op2` contains message `sd1_m1`.

2.3.2 Interaction Semantics

According to Micskei and Waeselynck [106], the sequence diagram semantics described in the UML superstructure provide only a basic idea of how the models work. However, these

semantics are ambiguous and incomplete; therefore, formal semantics need to be used to be able to understand how sequence diagrams are interpreted in practice.

Formal semantics theory support makes it possible to verify system designs. A complete definition of a formal modelling language would consist of a description of its well-defined syntax and semantics that enhance the readability and the expressiveness of the language. There is an increasing acceptance that formal methods form an essential part of the design of any reliable complex software system [106]. This is due to formal methods having the potential to illustrate ambiguities and design faults in order to avoid associated system failures. In particular, the formal model of a system can be used to prove system properties, such as performance, reachability, consistency and correctness mathematically. Moreover, formal models and methods make software designs more tangible by allowing rigorous validation and verification. Validation provides assurance that the design specifies the right system, whereas verification ensures that the end system satisfies the specifications [105]. Currently, many semantics are proposed for UML sequence diagrams. In this section, 10 approaches have been selected, listed in Table 2.2. In fact, there are more than 10 approaches proposing semantics for UML sequence diagrams but it would be impossible to include all of them. The 10 approaches selected are common semantics of the sequence diagrams that have influenced most of the others.

Störrle [144, 145] was one of the first to propose sequence diagram semantics in this area. His first approach [145] proposed semantics for plain sequence diagrams without CombinedFragments and later extended to cover semantics of most CombinedFragment operators, such as *Alt, par, opt* and so on. However, Micskei and Waeselynck [106] argue that there are some issues surrounding these. One of them is that the semantics focus more on CombinedFragments, whereas some of the ordering problems of basic interactions are not addressed. A similar approach is defined by Cengarle and Knapp [26, 27]. These semantics mainly focus on the interpretations of positive and negative traces in the sequence diagrams.

Harel and Maoz [78] argue that the operators assert and neg are insufficient for specifying forbidden behaviours. Thus, they propose a Modal Sequence Diagrams (MSD), this being an extension to sequence diagram, which adapts Live Sequence Charts (LSC) to the notation of

Table 2.2: Selected semantics

Name	Reference
Küster-Filipe	[54]
Knapp & Wuttke	[93]
Cavarra & Filipe	[24, 25]
Störrle	[144, 145]
Harel & Maoz	[78]
Fernandes et al.	[53]
Hammal	[74]
Eichner et al.	[49]
STAIRS	[79, 80, 102, 136]
Cengarle & Knapp	[26, 27]

UML. LSC extends MSC by allowing the elements of a Sequence Diagram to be specified as either mandatory (hot), or possible (cold) scenarios. Due to the sequence diagram not having a clear definition of the modalities of the diagrams, they used the model, LSC to Sequence Diagrams. However, there are some issues with these semantics. For example, the transition of the automaton is labelled only with the message name, but does not include information in the lifelines the message is sent or received from.

Furthermore, Steps to Analyze Interactions with Refinement Semantics (STAIRS) [79, 80, 102, 136] represent trace-based semantics for sequence diagrams. These semantics focus on a precise definition of refinement for Interactions. This approach is very similar to the one presented Störrle in [145].

Cavarra and Filipe [24, 25] also introduce semantics for sequence diagrams inspired by LSC. These semantics, using OCL, express 'liveness' properties in sequence diagrams, based on the results of LSC. This approach resembles the semantics proposed by Harel and Maoz [78], in

terms of specifying the mandatory and possible scenarios which referred to in this approach as 'may' and 'must' behaviour.

Further semantics are also presented by Knapp and Wuttke [93]. These authors used Interaction automata to represent traces of executions. Furthermore, they define some restrictions to how the problems of sequence diagrams can be overcome, such as replacing neg with a binary logic variant *not* is restricted to basic interactions. Moreover, these semantics do not clearly represent an interpretation of alternative CombinedFragment.

Fernandes et al. [53], propose a translation that produces coloured Petri Nets from UML use cases and sequence diagrams. This approach considers the behaviour of weak sequence diagrams, in addition to CombinedFragment operators (par, alt, opt, loop). Similarly, Eichner et al. [49] propose semantics for sequence diagrams based on coloured, high level Petri Nets. These semantics focus on basic constructs of the sequence diagrams, such as the start of a lifeline or sending and receiving a message.

Hammal [74] defines formal semantics for sequence diagrams, using a branching time structure rather than traces. This model (a lattice-like graph) represents traces of all sequence diagrams components, together with possible execution and can be directly unfolded into a transition system that captures the intended behaviour. It also proposes a method of extracting time properties from sequence diagrams and adding them into the graph for performance analysis.

Finally Küster-Filipe [54] present true-concurrent semantics, based on Labelled Event Structures (LESs). LESs are highly suitable for describing how traces of execution in sequence diagrams are able to capture the available notions, such as sequential, parallel and iterative behaviour. Moreover, the LES model takes into account the possible nesting of CombinedFragments and gives a very clear definition of the predecessors of each event. In the present thesis, the semantics of sequence diagrams are undertaken via an LES. An LES is chosen due to its simplicity for modelling traces of execution. Moreover, it supports the important operators of the sequence diagram forming the focus of this thesis. This LES are explained in greater depth in the following section.

2.3.2.1 Labelled Event Structures (LES)

Several possible semantics for sequence diagrams have been defined, as mentioned in the previous section. In the present thesis, the semantics defined in [54] are used, which introduces a very simple and intuitive behavioural model to capture interactions and contains the only true concurrent semantics available for sequence diagrams.

Prime event structures [138], or 'event structures' for short, describe distributed computations as event occurrences together with binary relations for expressing causal dependency (called *causality*) and nondeterminism (termed *conflict*). The causality relation implies (partial) order among event occurrences, while the conflict relation expresses how the occurrence of certain events excludes the occurrence of others. From the two relations defined for a set of events, a further relation is derived: the concurrency relation co. Two events are considered concurrent if and only if they are completely unrelated (i.e. neither related by causality nor by conflict). Please note that the following definitions (1, 2, 3 and 4) have been borrowed from the semantic presented in [54].

Definition 1. *An* event structure *is a triple* $E = (Ev, \rightarrow^*, \#)$ *where* Ev *is a set of events and* $\rightarrow^*, \# \subseteq Ev \times Ev$ *are binary relations called* causality *and* conflict*, respectively. Causality* $\rightarrow^*$ *is a partial order. Conflict* $\#$ *is symmetric and irreflexive, and propagates over causality, i.e.,* $e\#e^{'} \rightarrow^* e^{''} \Rightarrow e\#e^{''}$ *for all* $e, e^{'}, e^{''} \in Ev$*. Two events* $e, e^{'} \in Ev$ *are* concurrent*,* $e \; co \; e^{'}$ *iff* $\neg(e \rightarrow^* e^{'} \vee e^{'} \rightarrow^* e \vee e\#e^{'})$.

Definition 2. *An* event structure $E = (Ev, \rightarrow^*, \#)$ *is a discrete event structure iff for every* $e \in Ev$*,* local configuration *of* e*,* $\downarrow e = \{e_n | e_n \rightarrow^* e\}$ *is finite.*

An event structure is said to be *discrete* if the set of previous occurrences of an event is finite, i.e., there are always only a limited number of causally related predecessors to an event, known as the local configuration of the event (written $\downarrow e$). A further motivation for this constraint is given by the fact that every execution has a starting point or configuration.

Immediate causality refers to events such as e_1, $e_2 \in Ev$ that are causal and have no other events occurring between them: if $e_1 \rightarrow^* e_2$ has an immediate causality relationship, then e_1

is the immediate predecessor of e_2, and e_2 is the immediate successor of e_1. Alternatively, this relation could also be written as $e_1 \rightarrow e_2$.

2.3.2.2 Translating UML Sequence Diagrams into Labelled Event Structures

This section illustrates the translation of sequence diagrams into an LES, defined in [54].

Definition 3. *A sequence diagram can be represented as a tuple* $SD = (I, Loc, Loc_{ini}, Mes, E)$*, where:*

- I is a set of instance identifiers corresponding to the lifeline in the diagram.
- Loc is the set of locations.
- Loc_{ini} is the set of initial locations such that $Loc_{ini} \subseteq Loc$.
- *Mes* is a set of message labels.
- E is a set of events where the triple (e_i, m, e_j) represents a message m sent from event e_i to e_j.

Definition 4. *Let* $E = (Ev, \rightarrow^*, \#)$ *be an event structure and* L *be an arbitrary set. A labelling function for* E *is a total function* $\mu : Ev \rightarrow L$ *for mapping each event into an element of the set* L*.*

The labelling function is necessary to establish a connection between the semantic model (event structure) and the syntactic model (here a sequence diagram). The labelling function used in this case is a partial function. Intuitively, each location marked along a lifeline of an object in a sequence diagram corresponds to at least one (and possibly more) event(s) in the LES.

The set of labels used could be the set of locations in a sequence diagram, but usually involves more concrete information on what the location represents: the initialisation of an object, sending/receiving a message, beginning/ending an interaction fragment, etc.

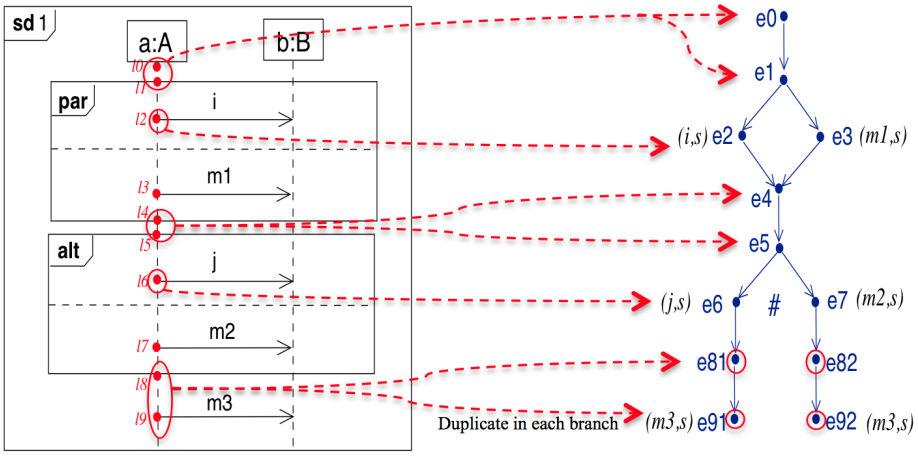


Figure 2.8: Event structure for object a of sd1

Consider the example in Figure 2.8. This figure illustrates the mapping between the sequence diagram and LES. The LES model shown in Figure 2.8 (right) has a direct correspondence to the locations of lifeline, a. Locations l_0 to l_7 correspond to events e_0 to e_7. Location l_8 is associated with events e_{81} and e_{82}. The graphical representation of the event structure E_a shows immediate causality between events (e.g. $e0 \rightarrow e1$) and direct conflict (e.g. $e_6 \# e_7$). The general causality relation can be inferred (e.g. $e_0 \rightarrow^* e_6$). By conflict propagation, $e_6 \# e_{82}$ is also implicated. Unrelated events are concurrent, such as events e_2 co e_3 where e_2 corresponds to sending i and e_3 to sending m_1. Intuitively, events e_1 and e_5 denote the beginning of the parallel and alternative fragments, respectively. Events e_{81} and e_{82} both correspond to location l_8 and indicate the end of the alternative fragment. These events must be in conflict because they represent different ways to reach the location. Note that there cannot be one end event in this case because conflict propagates over causality and would lead to an event in conflict with itself and hence an invalid event structure (conflict is irreflexive and propagates over causality).

In order to represent the LES of the complete sequence diagram, the semantics of LES were extended. Let I denote the set of objects involved in the interaction described by sequence diagram SD and Mes the set of asynchronous messages exchanged. Let the set of labels L be given by $L = \{(m, s), (m, r) | m \in Mes\}$. An event with the label (m, s) corresponds to the sending of message m, whereas an event with the label (m, r) indicates the receipt of message m.

Definition 5. *A model $M_{SD} = (E, \mu)$ for a sequence diagram SD is obtained by composition of the models $M_a = (E_a, \mu_a)$ of each lifeline instance $a \in I$. In M_{SD}, the set of events is given by $Ev = \bigcup_{a \in I} Ev_a$, and event labels are as before, that is, $\mu(e) = \mu_a(e)$ for $e \in Ev_a$. Let m be a message sent between lifeline a and lifeline b, and let $E_1 \subseteq Ev_a$ with $\mu_a(e_1) = (m, s)$ for all $e_1 \in E_1$, and $E_2 \subseteq Ev_b$ with $\mu_b(e_2) = (m, r)$ for all $e_2 \in E_2$. Then necessarily $|E_1| = |E_2|$ and for each $e_1 \in E_1$ there is a unique $e_2 \in E_2$ for each e_1 such that $e_1 \rightarrow e_2$ and local conflict $\#_a$ propagates over $\rightarrow$ to obtain conflict $\#$ in M.*

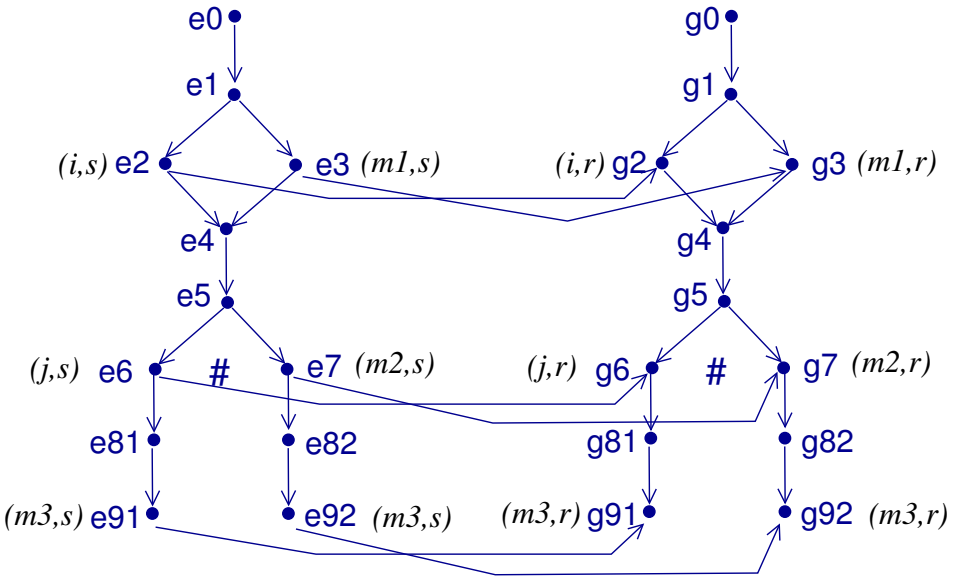


Figure 2.9: Model for sequence diagram sd1.

The overall event structure model for the diagram from Figure 2.5 is given in Figure 2.9. Conflict propagation is not shown explicitly but is as expected and propagates over the new causality relations gained from communication. For example, since $e_7 \rightarrow g_7$ by conflict propagation we also have $e_6 \# g_7$.

Definition 6. *Let $M_{SD} = (E, \mu)$ be a model for sequence diagram SD where $E = (Ev, \rightarrow^*, \#)$ is an event structure. A subset of events $C \subseteq Ev$ is a* configuration *in E iff it is both 1)* conflict free*: for all $e, e' \in C, \neg(e \# e')$ and 2)* downwards closed*: for any $e \in C$ and $e' \in Ev$, if $e' \rightarrow^* e$ then $e' \in C$. A maximal configuration denotes a* trace.

For example, the following is a trace for Figure 2.9: $C = \{e_0, e_1, e_2, e_3, e_4, e_5, e_7, e_{82}, e_{92}, g_0,$

$g_1, g_2, g_3, g_4, g_5, g_7, g_{82}, g_{92}\}$ which denotes the occurrence of m_2 and not j. More details on the semantics of sequence diagrams using LES can be found in [54].

2.4 Model Composition

Modern systems play a significant role in many aspects of human life, such as in health, economics, finance, education, communications and transportation. However, designing and implementing such systems is a very complex process, requiring engineers to make use of multiple models for expressing various scenarios and viewpoints. However, this separation helps to simplify the process of designing, managing and improving these systems. Nevertheless, this separation also requires an integration or recomposition step, in order to obtain a global representation of a system under construction and to reason about the system as a whole for the purpose of verification, validation and consistency [34]. For these reasons, model composition has become a significant and challenging step in the process of modern system development.

As stated earlier, manual composition can be carried out for a small system, but it is very difficult with larger tasks, because it is error-prone and time-consuming. As a result, automated model composition is vital to help designers recombine models into consistent views of systems under design/development. In recent years, automated model integration has received considerable attention [11, 20, 22, 57, 73, 91, 100, 133, 133, 135, 151, 154, 157, 159]. Current studies show that there are three operators used for model integration (merge, weave and composition), which are often mixed up in the literature [29, 34].

Model merge usually refers to building a global view of a set of overlapping models that consist of the same or related elements. The same elements indicate that the overlapping elements in the input models have the same structure and semantics. On the other hand, the related elements mean that the set of elements have a similar strategy but might be different in their structure or semantics. The overlapping elements are used in model merging techniques as a join point to combine the input models by unifying these overlaps [112].

In addition to the above, the *model weaving* used in Aspect-Oriented Modelling (AOM)

[57] to compose a set of cross cutting concern (i.e. aspects) into the $base$ model. Moreover, AOM provides mechanisms for separating crosscutting concerns in design models, and an intended change can be captured as an aspect. The original model is the base model, whereas the (possibly new) functionality that is required in several places is the aspect. Aspects are also particularly useful for dealing with non-functional properties and dependability concerns (including security, reliability, availability, safety and so on), which usually impact the system as a whole [133]. It is important to understand what an aspect will do and where and when it will affect the base model. Many AOM techniques use the term $advice$ for the action an aspect will take and $pointcut$ to specify more general rules of where to apply an advice. To analyse the effect of an aspect on a base model, the integrated system model must also be considered. This can be obtained by weaving the base and advice models in accordance to a pointcut.

Finally, *model composition* focuses on all activities that enable the building of a system from the union of several independent and dependent software artefacts. The term $composition$ comprises the usual terms of merging, weaving and union, as well as any activity whose intent is to create software from reusable parts of other systems [34]. Furthermore, model composition is the process of manipulating model elements from at least two source models, in order to produce a unified representation that may be serialised or only made available at run-time. In this thesis, the term 'composition' will be used as a default term for model integration, as it comprises all operators. However, a specific term will sometimes be used for a special case. For example, the term 'weaving' will refer to the composition of aspects.
Generally, in order to compose the software models, two fundamental conditions must be satisfied:

- Matching elements must indicate correspondence between equivalent elements of the source. The purpose of matching is to uncover how two models correspond to each other. Moreover, the process of matching defines matching criteria that identify common elements of the input models. The matching criteria are based on identifying properties of the source model elements and comparing them, such as matching the element names.

- The composing of equivalent elements identified earlier to produce a composed version

of the models.

These conditions are essential in the model composition process, which must be considered in all composition approaches, in order to be able to generate a single coherent model that will represent a global view of the system. Model composition techniques can be divided into two aspects: composition techniques for static models and composition techniques for dynamic models.

2.4.1 Static Model Composition

The problem of static models composition has been studied in many domains, such as the databases entity diagram [132], class diagram composition [13, 56, 67, 98, 129, 135, 158, 159], and various other system varieties. In static composition, a common matching criterion between the model elements is based on their names. For example, if two classes have the same name, they can be composed together. This means that matched classes are combined into one and their properties represented by model elements, such as attributes, that match only once will appear in the merged model. Finally, properties that do not have any correspondence in the other model will be added to the composed model. The above procedure is applied in most static composition approaches.

Zhang et al. [159] and Rubin et al. [135] have used Alloy for the composition of class diagrams. Firstly, they transformed the class diagrams into Alloy language and then composed them using certain logical constraints that defined how the classes should be composed together. The composition performed in this approach was based on matching the names of elements. Although the transformation was carried out with clarity, the composition process is not clear and seems to have been performed for a specific instead of a generic case. The above approaches do not, however, have a supporting tool to automate the transformation.

Morin et al. [129] propose a tool called SmartAdapters. This approach represents a model weaver that supports variability. It has been designed to provide capabilities for functional concerns to be reused in the context of variability. According to Clavreul [34], SmartAdapters involve a homogeneous and asymmetric approach to weaving reusable concerns (i.e. aspects)

into one or more base models. Each aspect is related to an adapter that declares a composition protocol. A composition protocol is a set of atomic operations and a set of target model elements that specify how an aspect should be woven with other aspects.

Another common tool for weaving is XWeave [67]. This is a tool for weaving (class diagrams) models encapsulated as crosscutting concerns into base models. Similar to other composition tools, XWeave takes a base model and one or more aspect models and weaves them together in a user controllable way. However, this tool cannot remove, change or override existing base model elements using aspects.

All of the above approaches have been used in composition techniques based on name-matching. However, France et al. [56] raise some problems that might occur with name-based matching. They argue that the composing models using a name-based matching are inadequate and may give rise to conflicts. For example, it could be the case that two classes with the same name might not represent the same concept, or may have conflicting properties. Thus, the use of name-based matching in the composition of static models could help reduce the occurrence of some naming conflicts, but will not be able to prevent them all. In their approach [56], France et al. suggest a signature-based composition technique that uses signatures instead of names to determine matching model elements. A signature consists of a set of information properties (attributes and association end) that provide enough information to detect any conflicts between models. The above researchers developed a tool called Kompose using Kemata [55] to support the composition technique. However, their work does not offer an explicit definition of the glue between the models. In other words, this approach does not explain how input models can be glued together (pointcut). A similar idea was presented by Pottinger and Bernstein [132], who proposed an algorithm designed to solve the problem of possible conflict in databases.

2.4.2 Dynamic Model Composition

The modelling of system behaviour can support developers in identifying behaviour flaws early on in the development process and significantly assist in meeting requirements and in design processes. Behaviour models focus on the semantic aspects of a system, rather than its static

aspects. Sequence diagrams, state machines and other behaviour diagrams are convenient for modelling system behaviour. However, one of the significant limitations of behaviour modelling is the complexity of building the models in the first place [148]. Therefore, to reduce this complexity, developers design them in steps. This process often results in partial specifications being captured in models, focusing on a subset of the system behaviour. To produce a model of the whole system, such partial models must be composed together, because this will help developers understand the overall behaviour of the system. Behaviour model composition has several advantages, such as reducing the complexity of the system by eliminating redundancies and discovering any gaps that could affect the system's security [6]. Another advantage is delivering an executable model early in the 'requirements' process, which will enable a wide range of validation analyses, such as simulations and consistency-checking techniques [148].

A behaviour model that results from the composition of partial specifications should illustrate all possible behaviours that do not violate the properties. This model represents all the behaviours that the system will provide once implemented. In other words, the final system cannot enact more behaviour than what is described by the composed model. Dynamic composition is an active area of research in various behaviour modelling, such as UML dynamic models [6, 10, 11, 20, 22, 57, 91, 100, 113, 133, 151, 154, 159], Petri Net [28, 43, 45, 51, 71, 72, 127, 149, 160, 160], and Business Process Modelling (BPM) [62, 66, 84, 96, 97].
In terms of dynamic models of UML, several studies have focused on state machines composition [10, 39, 113, 152]. Nejati et al. [113] presented an approach for merging state machines that exploits syntactical as well as semantical information provided by the models to compare variants and perform consistency checks. The correctness of the result in this work is based on the definition and the algorithms that are created. However, while this approach seems suitable for simple models, it is not clear how to apply it to models with complex operators, such as choice or parallel behaviour.

Whittle and Jayaraman [152] also studied the composition of hierarchical state machines from UML 2.0 interaction overview diagrams, which contain activity diagrams constructed to specify complex behaviour. The generated hierarchical state machines are used to simulate

scenarios and improve readability.

Another common approach in industry, especially in the domain of telecommunications and avionics, is the Motorola WEAVR; a tool that considers the systems actions as state machines [39]. State machines are used to model the advice, pointcut and base models, and they use an 'around' operator in aspect-modelling to weave advice into the base model. According to [10], the Motorola WEAVR is the first commercially available aspect-modelling tool that focuses only on state machines.

The challenges of model composition have also been studied using Petri Net, which is referred to as synthesis. Petri Net is a formal modelling language often used to model control flow in a system. It is capable of modelling complex behavioural properties, such as conflicts (choice) and concurrencies (parallelism) [110]. Many synthesis algorithms and techniques have been used for different types of synthesis, such as top-down [149], bottom-up [43], hybrid [160], the knitting technique [28, 161], reduction [51, 71] and rough set [127].

The composition also has been studied in the Business Process Model (BPM)[30]. BPM is defined as a mechanism for describing and communicating with the current or intended future state of a business process[134]. BPM is capable of modelling complex behaviour of the systems. Various studies have investigated the composition of the BPM [62, 66, 84, 96, 97].

The composition study in the present research focuses on sequence diagrams. The following section will investigate several studies on sequence diagram composition.

2.4.2.1 Composition in Sequence Diagrams

As mentioned above sequence diagram composition has already been studied by some researchers. For example, Widl et al. [154] present an approach to sequence diagram composition using their corresponding state machines. The composition in this approach was performed formally, using a SAT solver called Picosat [18]. This method also presents a prototype tool, which automatically generates SAT encoding and represents the diagrams. However, this approach does not support the composition of sequence diagrams with CombinedFragment operators.

On the other hand, Liang et al. [100] present a method for integrating sequence diagrams,

based on the formalisation of the sequence diagram into particular kinds of typed graphs, called SD-graphs. The idea presented in their publication is designed for a sequence diagram consisting of lifelines and messages. However, this approach is similar to that of Widl et al.[154], which does not support the composition of sequence diagrams with complex behaviours, such as parallelisms and alternatives.

Bowles [20] also presents an approach to sequence diagram composition. Here, the author maps sequence diagrams into an LES and composes LES models by injecting new behaviour into a model through a category-theory based construction. This approach illustrates its ability to compose sequence diagrams containing, for example, alternative and parallel Combined-Fragments, However, the processes of transformation and composition have been performed manually.

Klein et al. [91] ppropose algorithms to weave Message Sequence Charts (MSC) by taking into account the semantics of these MSCs. The composition is specified at modelling level using time automata. According to Clavreul [34], this approach is not designed to be generic, due to the definition of pointcut and advice and the proposed algorithms are also specific to the MSC structure. Furthermore, this approach does not have a tool to automatically conduct the weaving and evaluation of the efficacy of the algorithms.

Another approach by Klein et al. [90] defines semantics-based sequence diagram aspects. In this approach, the above authors propose four match strategies: strict part, general part, safe part and enclosed part. These interpretations describe the degree of strictness when trying to detect a set of model elements that are related to each other. The two strictest match strategies (strict part, enclosed part) show that they cannot be inserted between the matched pointcut events on a lifeline for any event which is not included in the pointcut. The second two match strategies (general part, safe part) are less strict, since they allow any event between the matched pointcut messages to be included in the woven model. A similar approach was proposed by Grnmo et al. [70]. This method presents a semantics-based technique for weaving behavioural aspects into sequence diagrams. These authors define lifeline-based weaving upon trace-based equivalence classes. In other words, their semantics performed aspects of matching and weaving at the level

of the lifelines and their matched events. Grnmo et al. also define the semantics of weaving models with unbounded loops. In subsequent work, Grnmo et al. [69] proposed a conformance issue for aspects and ensured that the woven outcome was always the same, regardless of the order in which the aspects were applied. Moreover, they offered semantics-based matching, which is a process that looks for matches in the semantics of the two diagrams.

Whittle and Jayaraman [10] however, introduced a tool called MATA. This tool uses graph trans- formations to specify and weave aspects based on sequence diagrams. This approach focuses on the structures of a sequence diagram in the weaving process, but puts less emphasis on the semantics of the composition. The advantage of weaving the structure of a sequence diagram is to preserve the original structure of the models. Nevertheless, weaving is not guaranteed [69]. In addition, Whittle et al. [151] presented a composition technique that creates model elements of sequence diagrams, even if these elements have different names when matching their rules. However, this approach still does not address any of the conflicts that might occur during composition.

Reddy et al. [133] propose the direct composition of sequence diagrams. Their approach uses UML sequence diagram templates for describing the behaviour of aspects designed and also tags for behaviour composition. In their work, an aspect may include position fragments (e.g. begin, end) to designate the location to be added in the sequence diagram.

In addition to the above, aspects can sometimes be used to model non-functional concerns, such as dependability requirements, which usually cut across several parts of a system. Regarding the use of AOM for security, [156] presents a method of analysing the performance effects of security properties specified as aspects. Moreover, Whittle et al. [153] used sequence diagrams to model and execute misuse case scenarios (both desired and attack scenarios) for secure system development. Mitigation scenarios were then designed as aspect scenarios and woven into the core behaviour, in order to prevent the execution of the attack scenarios.

The composition technique is also used in multi-view modelling [89]. Multi-view is a methodology that allows a developer to describe a software system from multiple points of view, such as structural and behavioural, using different modelling notations. The composi-

tion of multi-view modelling is a significant process as it creates overall views for debugging, simulation or code generation purposes and also performs consistency checks during the composition. Kienzle et al. [88, 89] presented an approach called Reusable Aspect Models (RAM) for modelling and weaving an Aspect-Oriented multi-view. Their approach integrates a class diagram (static view), sequence diagram and state diagram (dynamic view). However, the matching and weaving processes are performed at the structure level of the sequence diagram. The authors plan to extend the approach by adding another type of view that describes the traces of execution of the behaviour models. Finally, Bowles and Bordbar [22] present a composition method that maps a design consisting of multiple views modelled (combination of class diagrams, sequence diagrams and OCL) into LES used for detecting and analysing inconsistencies. The composition process is performed at the level of the LES. However, this approach does not show the parallelisms that may occur after the composition; the processes of transformation and composition have been performed manually.

Since the composition of this thesis focuses on sequence diagrams in particular, Table 2.3 shows some of the existing composition approaches.

As illustrated in this section, most existing approaches have been performed manually or use algorithms to compose simple sequence diagrams. However, the idea of the composition in this thesis is to automate the composition of sequence diagrams using constraint solvers. The following section provides some background on the constraint solvers used here.

2.5 Constraint Solvers

Constraint solvers, such as SAT/SMT solvers are very common in the area of formal verification and many approaches have been developed on utilising SAT/SMT solvers to verify models or programs [12, 36, 63, 101]. Among them, Alloy is one of the most popular in the research on SAT solvers [82]. Alloy, a model finder, is a well-designed tool that can be used for finding instances of a model in a finite search scope [75, 81, 82]. Unlike other model-finding tools, Alloy uses first-order relational logic to describe a model. Nowadays, Alloy receives considerable

Table 2.3: Selected approaches to composing sequence diagrams

Approach	Notation		Support Combined-Fragment	Tool support
	Source	Target		
Reddy et al. [133]	SD	SD	Yes	Yes
Whittle et al. [10]	SD	State machine	Yes	Yes (MATA)
Ameedeen et al. [7]	SD	PN	Yes	Yes (SD2PN)
Widl et al. [154]	SD	State machine	No	Yes
Liang et al. [100]	SD	Typed graphs SD-graphs	No	Yes
Bowles [20]	SD	LES	Yes	No
Klein et al. [91]	MSC	Automata	No	No
Klein et al. [90]	SD	SD	No	Yes
Grnmo et al. [70]	SD	SD	Yes	Yes

attention in model analysis and composition [8, 9, 58, 95, 104, 139]. For example, Anastasakis et al. [8] propose a common approach called UML2Alloy, which focuses on the transformation between UML class diagrams and the Alloy language. In their approach, they present a list of rules which can map a UML class diagram and OCL constraints to the Alloy language. In addition, Rubin et al. [135] and Zhang et al. [159] propose composition approaches that use Alloy for class diagrams. Due to the popularity of Alloy for analysing UML models, it is the natural choice for representing and composing behaviour models, such as sequence diagrams. The following section will therefore explain Alloy in greater depth.

2.5.1 Alloy

Alloy is a modelling language based on first-order relational logic, which was developed by a software design group at the Massachusetts Institute of Technology (MIT) [118]. It is commonly used for modelling object-oriented systems. Alloy is roughly a subset of the notation

of Z [142]. Data domains in an Alloy model are defined using signatures (keyword: sig) and are represented as sets. A signature may extend another signature, in which case the domain defined by the first is a subset of the domain of the extended signature. A signature that is declared independently of any other is called a top-level signature. Extensions of a signature are mutually disjoint, as are top-level signatures. A signature can also be *abstract*, in which case its domain only contains elements that belong to its extending signatures. In addition, signatures may also contain *fields*, which are captured by relations. Axioms in Alloy are called *facts*, which can be given a name. These must hold at any time. *Predicate* is a constraint that only holds when invocated, which always has a name. Alloy formulae often use the atomic predicate in (inclusion), standard connectives from first-order logic, and for the quantifiers all (universal) and $some$ (existential).

Figure 2.10 depicts a simplified metamodel of Alloy. This thesis only considered some of the Alloy language features, which are explained fully by Jackson in [81].

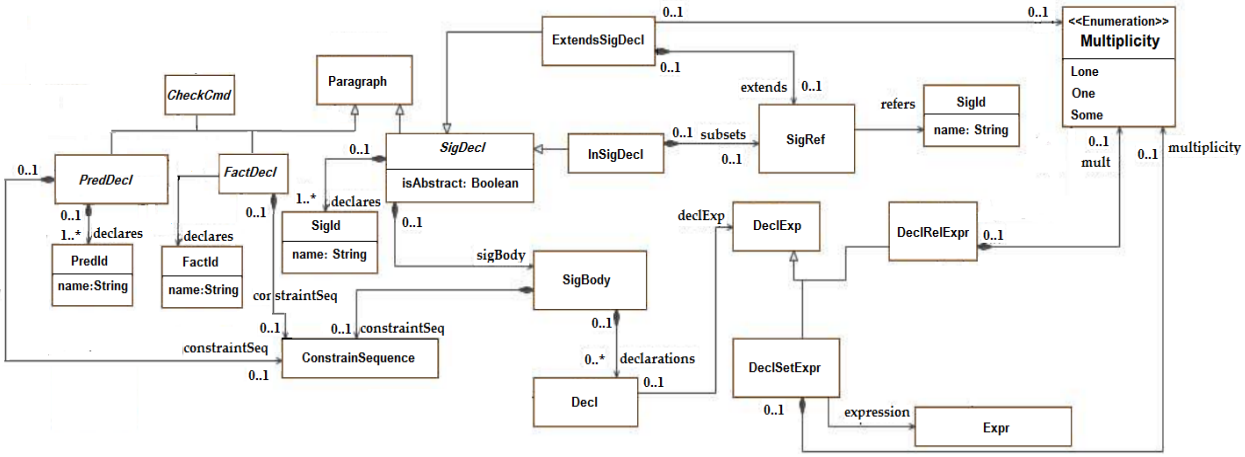


Figure 2.10: A subset of an Alloy metamodel [9]

Alloy uses a tool called Alloy Analyzer, a fully automated constraint-solver tool developed for analysing models written in the Alloy language. The analyser offers multiple functionalities: simulation using (run command) and assertion using ($assert$ and $check$ commands). The purpose of the simulation is to produce a random instance, which represents the running of the model and thereby confirms the specification. An assertion is a constraint that needs to be satisfied and should follow from the model facts. The Alloy Analyzer works by translating

Alloy formulas into Boolean expressions with the help of the KodKod [146] model finder. The Boolean expression is then automatically analysed using SAT solvers (i.e. SAT4J [17], ZCha [103] and MiniSAT [48]), and the solution is then displayed to the user as a graph (instance). The Alloy Analyzer is designed to perform finite scope checks, even on infinite models. A scope is a positive integer number, which specifies the number of instances of each model element in an instance of the system that is being analysed by the solver. Consequently, the user can specify the model elements scope to limit the domain. The default number of Alloy scopes is three atoms of every model element, unless the user changes the scope. For more details on the notion of scope, please refer to [[81], Sect. 5].

Alloy offers another valuable feature, which is to support the user in debugging conflicts between logical statements of models. This feature is also known as an UnSAT Core [81]. Therefore, if the Alloy Analyzer cannot find an instance conforming to the model, UnSAT Core highlights the conflicting statements that make the model $unsat$.

```
1.    abstract sig Person {
2.       children: set Person,
3.       siblings: set Person}

4.    one sig Man extends Person {}

5.    one sig Woman extends Person {}

6.    sig Married in Person {
7.       spouse: one Married }

8.    fun parents [] : Person -> Person { ~children }

9.    fact {no p: Person | p in p.^parents}

10.   run {} 3

11.   assert ChildrenArenotParents {
12.       all p: Person | no p.children & p.parents }

13.   check ChildrenArenotParents for 3
```

Figure 2.11: A sample of an Alloy model

Figure 2.11 shows a small Alloy model adopted from [61], which presents the common

features of Alloy. An $abstract$ signature (line 1) declares a domain called *Person*. This means that the abstract signature *Person* only contains elements that belong to its extensions. Lines 2 and 3 show the fields of the abstract signature *Person* (*children* and *siblings*). The *set* keyword in the fields corresponds to an association with a set of *children* and *siblings*, which means that there can be either zero or any number of *Persons* related to *Person* through the relations *children* and *siblings*. The signature in line 4 declares a signature called *Man*. This signature is extended from the abstract signature *Person*, which means a subset of the *Person* set. The keyword *one* in the *Man* declaration indicates that there is exactly one instance of the signature (unique set). Lines 5 and 6 are similar to the signatures declaration that has already been explained. A *function* and *fact* in lines 8 and 9 define the constraints of the parents relation. The constraints declare that no *Person* can be his or her own ancestor. Line 10 shows an empty run command, which will produce a random instance of the model and simulates the model. The Alloy Analyzer in this example will produce an instance of the model (solution), using three atoms at most for each of the signatures declared in the model due to the default scope of 3, as previously mentioned. Thus, the instance of this example will use a scope of three just for the *Married* signature, since the *Man* and *Woman* signatures are defined as singletons (*one* keyword). An assertion in line 11 defines that no person has parents that are also children at the same time. Finally, a $check$ command in line 13 asks the analyser to confirm the assertion for a scope of three.

2.5.2 SAT Solver

The SAT solver was mentioned in the Alloy section; it is the backbone of the Alloy Analyzer, which solves the Boolean expressions produced by Alloy Analyzer and returns a solution. This section will give a brief background about the SAT problem. A Boolean satisfiability problem (SAT) is a decision problem. Given theory T, which contains a set of propositional Boolean logical formulas $\{S_1, S_2,...,S_n\}$, the aim of the SAT solver is to find an assignment of variables that satisfies every propositional formula [75]. If the assignment exists, a solution is found. A computational procedure, which could decide if the set of boolean formulas is satisfied/un-

satisfied is called a *decision procedure*. However, the computation of such an procedure is not easy. The SAT problem has been proven to be the first known nondeterministic polynomial time complete (NP-Complete) problem [35].

$$(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_3 \vee x_4) \wedge (\neg x_1 \vee \neg x_2 \vee x_4)$$

Figure 2.12: A simple example of propositional logic formula in CNF

Figure 2.12 illustrates a simple example of an SAT problem, which is created in conjunctive normal form (CNF) [130]. The variables x_1, x_2, x_3 and x_4 in this formula are called literals, and each sub-formula$(x_1 \vee \neg x_2 \vee \neg x_3)$ is called a clause. To solve boolean satisfiability problems, many SAT solvers have been implemented to automatically solve Boolean satisfiability problems [17, 48, 64, 103]. These solvers were developed to answer a large number of Boolean formulas and determine their satisfiability. If the formulas are satisfied (sat), an assignment of each variable is returned. Otherwise, the SAT solver returns $unsat$. Modern SAT solvers are extremely efficient and designed to solve up to one million clauses in a very short time (within few seconds) using the Davis-Putnam-Logemann-Loveland (DPLL) algorithm [40]. According to Hao [75], the DPLL algorithm is bounded by exponential time (EXP), which means that in the worst case it is very slow. However, in most cases, the algorithm is fast and returns the solution within a few seconds.

Solving the formula in Figure 2.12 is very straightforward. One of the possible assignment for this formula is x_1 = $true$, x_2 = $false$, x_3 = $true$ and x_4 = $true$. SAT solvers are powerful tools that can be used to solve a difficult problem by translating that problem into SAT encoding. However, the difficulty is that such a translation is difficult since a vast number of propositional logic formulas are needed to express the problem. Encoding using plain Boolean has some limitations, including that it does not directly allow integer encoding. Therefore, SMT was proposed to provide theories to express such problems without losing completeness and automation [75].

2.5.3 SMT Solver

The Satisfiability Modulo Theories (SMT) problem is also a decision problem, just like the Boolean SAT problem. The expression of an SMT problem can be performed through a combination of many theories provided by SMT solvers. For instance, SMT solvers can combine an integer theory and a quantified Boolean theory to specify a problem [16]. Due to this support from different theories, the translation of a problem to SMT could be easy compared to translating it into a SAT problem. The main difference between SAT and SMT solvers is that SMT solvers accept systems in an arbitrary format, while SAT solvers are limited to Boolean equations in CNF form. This means that SMT solvers are supported by rich background theories, such as linear integer arithmetic theory, which is defined in the Satisfiability Modulo Theories Library (SMT-Lib)[16], while SAT solvers only use propositional logic. A typical example is solving a linear integer arithmetic equation, where the input is a set of equations written in human readable format and the output is the assignment for each variable in the equations[75].

SMT problems are a very active area of research, which has received considerable attention. Many SMT solvers are being developed [19, 42, 44, 47]. Z3 is state-of-the-art SMT solver that has been used in this thesis and was developed by Microsoft Research. It is a high-performance solver targeted at solving problems arising in software verification and software analysis [42]. Like any other SMT solver, Z3 supports many types of declarations, such as Integer, Real and Boolean, as well as allowing users to declare new sorts (types).

Functions:

Functions in Z3 are the basic building blocks of SMT formulas. Unlike programming languages, where functions have side-effects, can throw exceptions, or never return, functions in classical first-order logic have no side-effects and are total [42].

Constants are functions that take no arguments, and $Const(a, A)$ is written to declare a constant a of type A.

Figure 2.13 shows a partial metamodel for Z3, which was defined in order to facilitate the transformation of the model in this study. Functions can represent a variable or mathematical

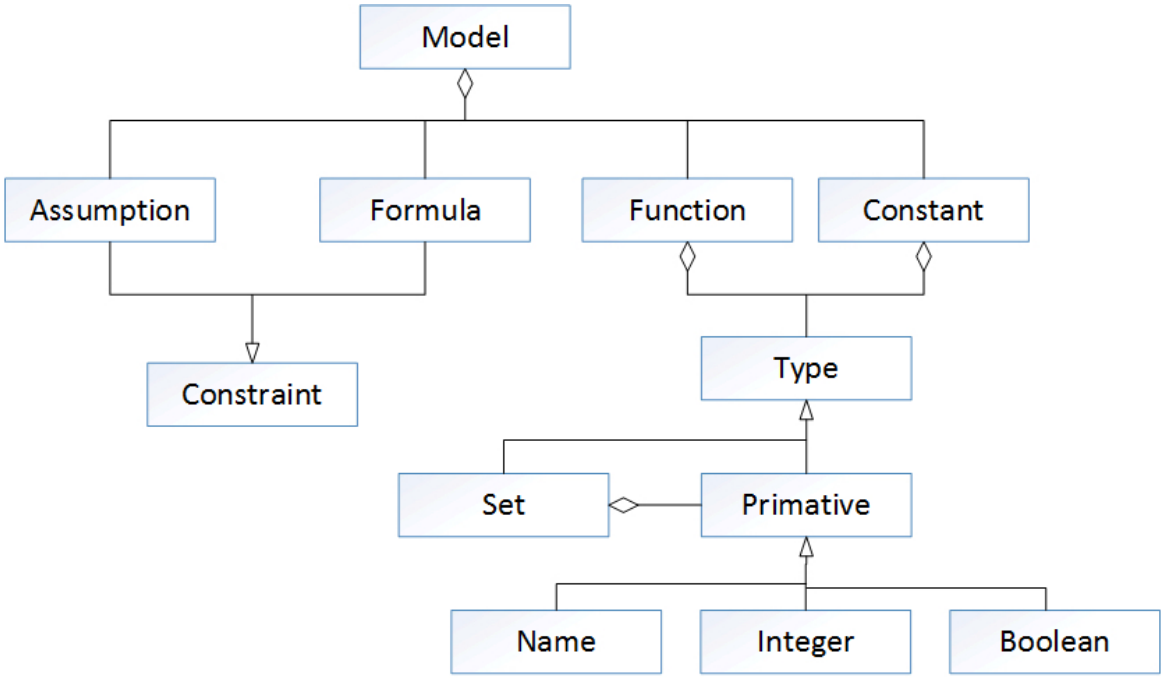


Figure 2.13: A Subset of Z3 metamodel

function. Moreover, a function can have one or more types restricted to primitive types (Integer, Name and Boolean) and additionally the type Set. The Name type allows uninterpreted functions to be used.

Boolean Logic:

Z3 supports Boolean operators, such as And, Or, Not, Implies (logical implication), and equality $==$ (used for bi-implication) among others. Universal ($ForAll$) and existential ($Exists$) quantifiers are also supported by Z3. In Z3, it is possible to create a general purpose solver using $Solver()$ and associate it with a particular variable by declaring $s = Solver()$. Later, constraints may be added to the solvers through the method $add()$. All the constraints associated with a solver can be checked (solved) by calling the method $check()$. The result of the procedure is similar to that of a SAT-Solver: either sat (satisfiable, a solution was found), or unsat (unsatisfiable, no solution exists). Finally, if the formulae are satisfied, a method $model()$ can be called to retrieve a textual representation of the solution. In this work, Z3 models were written in Python using Python API for simplicity of interpretation and to parse the solution into a graph.

In Figure 2.14, constants x and y are declared as an integer variable in Z3, named x, y. Z3,

```
1. x = Int('x')
2. y = Int('y')
3. s = Solver()
4. s.add(x > 2, y < 10, x + 2*y == 7)
5. s.check()
6. s.model()
```

Figure 2.14: A simple Z3 model

like Python, uses '=' for assignment and Line 4 shows the Z3 constraint ($x + 2^{*}y == 7$). This constraint was checked with the Z3 solver using the command $s.check()$. After this checking, Z3 shows one of the possible solutions using the command s.model: that is, $x = 7, y = 0$. The next section presents current approaches use constraint solvers for model composition.

2.6 Model Composition via Constraint Solvers

In recent times, constraint solvers have been widely used for the fully automated composition of static and dynamic UML models. Rubin et al. [135] used Alloy to compose class diagrams based on the syntactic properties of metamodels. As mentioned earlier, they converted the class diagrams and the metamodels into Alloy and composed them based on a set of logical constraints. The matching in this approach is named-based matching. Although the composition model is a union of two input models, the instance of the composed model displays all the elements of the input models, including the matched elements, as well as which one of them should occur. In addition, the composition glue is not generic but seems to be designed for a specific case instead. In fact, this approach is acceptable for small examples, but in complex examples, the duplicated elements will make it difficult to validate the analysis of the final composition.

Zhang et al. [159] presented a weaving-based model composition framework (WMCF). This work is based on the work of Rubin et al. [135], but the authors added the weaving process and evaluated the approach by composing a simple example. Although these approaches use Alloy

to compose UML static models, they are not focused on the composition of UML behaviour models, such as sequence diagrams in Alloy.

Alloy was also used by Mostefaoui and Vachon [109] to analyse and weave behavioural interactions of aspect-oriented models. A base model is transformed into Alloy as well as into pointcut specifications and advice. In this work, multiple aspects are composed into a single Alloy entity and then woven with the base behaviour using the operators before, after, or both before and after. However, this work did not consider the operator *around* in the weaving. For example, if one or more advice elements might have matched one or more elements in the base model. A similar approach was proposed by Nakajima and Tamai [111], who used Alloy to automatically weave aspect-oriented models. However, they did not address the rules for transforming origin models into Alloy.

In addition, Widl et al. [154] presented an approach for composing a sequence diagram via the Picosat solver. This approach implemented a tool, which automatically translates the diagram into Picosat encoding. The solver then processes the composition and returns the results, which are subsequently interpreted by the tool for analyses. However, this approach does not support sequence diagrams with CombinedFragment.

The current thesis proposes a fully automated composition technique using both Alloy and Z3- SMT constraint solvers. This approach has the ability to compose complex behaviour models, such as sequence diagram containing CombinedFragments like *alt* and *par*. The composition considers both aspects of the model, i.e. the abstract syntax (static representation) and traces of execution (semantics), when the models are composed to produce a correct solution. The following chapters will explain the idea of the composition in greater depth.

2.7 Model Driven Architecture (MDA)

Model driven development (MDD) techniques aim to enhance the role of modelling in software development [143]. These techniques allow the developer to model the required functionality and the overall architecture of the system instead of calling on developers to spell out every

detail of the systems implementation using a programming language. Hence, MDD results in reduced development cycles and a lower cost of software production.

To ensure that the methods designed can be adopted by the software industry, it is crucial to follow standards set by the model driven architecture (MDA) framework [92]. MDA is a framework for software development that was proposed by the OMG. It provides a set of guidelines for the structuring of models and their specifications. It also defines a standard for application design and implementation.

Central to MDA is the notion of metamodels [65]. A metamodel defines all elements that are available for a designer to use when modelling with a language. In MDA, a model transformation is defined by mapping the meta-elements; the constructs of the source metamodel (e.g. sequence diagrams) are mapped onto constructs in the target metamodel (e.g. Alloy), as shown in Figure 2.15. Subsequently, every model arising from the source metamodel can be automatically transformed to an instance of the destination metamodel with the help of a model transformation framework. The source and target metamodels are specified using a common language called meta object facility (MOF) [115], while the models are instances of metamodels.

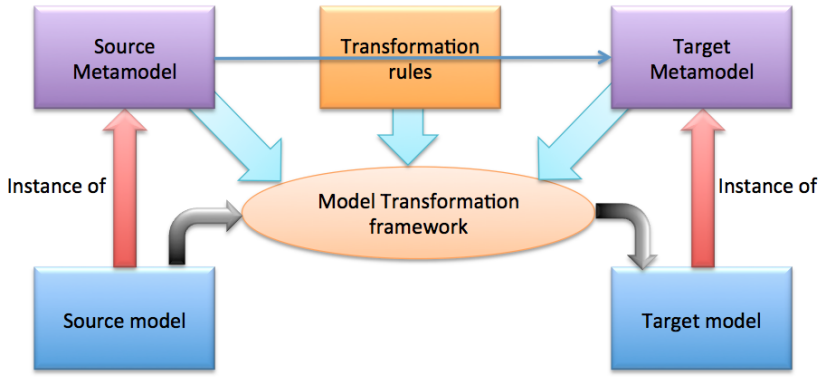


Figure 2.15: MDA outline

Figure 2.15 illustrates an outline of the MDA and the process of model transformation. To transform any instance of the source metamodel, the model transformation framework executes the rules for creating an instance of the destination metamodel, in addition to defining how various elements of one metamodel are mapped into the elements of another. The process of

model transformation is carried out automatically via the tools and is commonly known as the model transformation framework. A typical model transformation framework requires the following inputs: the source and target metamodel and the number of transformation rules.

Currently, there are many tools to support model transformation and these have been developed in both academia and industry, such as Kermeta [122], Arcstyler [119], OptimalJ [123], Xactium [2, 31] and ATLAS [85–87]. Despite the fact that these tools allow the specification and implementation of a model transformation and therefore provide a rich set of functionalities, such tools are inherently complex. In particular, for scholars in academia or research laboratories, who are only interested in experimentation and the creation of prototypes, the steep learning curve is a significant hurdle. Thus, this thesis makes use of Simple Transformer (SiTra) [3, 125], which offers a minimal framework for the execution of transformations to implement the transformation rules.

2.8 Simple Transformer (SiTra)

SiTra is a simple Java library implemented at Birmingham University by Akehurst et al. [3]. It is developed to support programming approaches to write transformations that intend to use Java for writing transformations. SiTra contains two main interfaces: the rule and transformer interfaces. The rule interface should be implemented for each transformation rule, whereas the transformer interface provides a framework for the methods that carry out transformations.

In SiTra, the developer is required to define the transformation rules by implementing the rule interface. The rule interface contains three methods: $check()$, $build()$ and $setProperties()$. If the rule is valid for the source element in question, the method $check()$ returns as true, and then the method $build()$ is executed.

The method $build()$ generates the target model element. Finally, $setProperties()$ is used to set the attributes and links of the new created target element. SiTra has been applied to model transformation in many application domains [4, 5, 9, 21, 23, 60]. For more details, please refer to [3, 125].

Traceability is a significant feature in model transformations, which have received considerable attention recently [52, 126, 139]. This feature is specifically used during the process of model transformation to keep track of which elements of the source model have been transformed into which elements of the destination model, and visa versa. For instance, after the source model has executed the model transformation, only the necessary elements of the target model can be updated by traceability links without having to execute the whole transformation again. For more details of traceability with SiTra, please refer to [3].

2.9 Chapter Summary

This chapter provides preliminary studies on UML in general and for sequence diagrams specifically. Following this, it provides background on LES; a type of formal modelling used to represent the semantics of sequence diagrams. Model composition has been studied with regard to different system aspects, such as static and dynamic composition. This is illustrated in section 2.3. However, most existing approaches have been performed manually or use algorithms. The work in the present study is close to the approaches proposed in section 2.5. However, these days, most strategies use constraint solvers to compose static models or a simple behaviour model. Therefore, this thesis presents a fully automated composition technique for creating a sequence diagram from partial specifications captured in multiple sequence diagrams, with the help of constraint solvers. In subsequent chapters, the methods for composing this model will be discussed in greater depth.

CHAPTER 3

EXACT METAMODEL RESTRICTION (EMR)

3.1 Overview

As noted in section 2.8, this work forms a discussion of a composition technique for behaviour models using constraint solvers. The aim of this research is to propose a fully automated composition technique for sequence diagrams and detect any inconsistencies arising during composition. The approach is as follows: firstly, Chapter 3 presents a model composition at the metamodel level, through the Exact Metamodel Restriction (EMR), outlining the methodology employed for the transformation and composition of sequence diagrams. In addition, there is a definition of formal composition semantics to guide the composition process. Secondly, Chapter 4 outlines transformation rules and the composition of sequence diagrams via Alloy. Thirdly, Chapter 5 discusses the second automatic composition of sequence diagrams via Z3-SMT. Finally, Chapter 6 compares the Alloy and Z3-SMT approaches presented.

The current chapter focusses on a technique for model composition at the metamodel level through EMR. A software model completely described by a set of logical constraints at the metamodel level. These models are composed through the union of logical constraints for each model, as well as constraints describing the composition glue. At the metamodel level, this gives the exact instance of the metamodel for the composition. In addition, there is a presentation of the formal semantics for composition, which guides the composition process.

This current chapter consists of the following sections: section 3.2 and sections 3.3 discuss the mechanism of EMR and the use of EMR for static models; section 3.4 outlines the compositions of static models via EMR; section 3.5 demonstrates the challenges in dynamic models; section 3.5.1 reveals the ways in which a sequence diagram can be described via EMR as an example of dynamic model; Section 3.5.2 illustrates the composition of sequence diagrams; Finally, in section 3.5.3 the composition is treated with LES.

3.2 Exact Metamodel Restrictions

As mentioned in section 2.2, metamodels represent all the involved model elements of a domain, including their relationship. Logical statements written in the context of metamodels play key roles, e.g. expressing well-definedness for the elements, the concept of equality between models' parts and so on. As the metamodel represents all compliant models, adding extra logical constraints can restrict the list of models compliant to a metamodel. Furthermore, it is possible to start from a given model, M, and exert enough logical constraint, $\mathcal{L}$=$\{\ \mathcal{L}_1, ..., \mathcal{L}_k\ \}$ on its metamodel, MM, such that the combination of MM with additional logical constraints, $\mathcal{L}$, can uniquely determine the original model, M. Thus, the pair $(MM, \mathcal{L})$ uniquely determines, M.

The process of identifying logical constraints $\mathcal{L}$, is referred to as Exact Metamodel Restrictions *EMR*, which can be used in an automated instantiation of the model via constraint solvers. For example, the use of the Alloy model finder for a given metamodel MM, along with a correct set of constraints, enables Alloy to be used to automatically recreate the model (see Figure 3.1).

The concept can be simplified as follows: assume that *John* is a student, and thus an instance of students metamodel. The metamodel contains many instances (i.e. students), such as Steve, Sarah, David, etc. However, these instances have different properties, i.e. date of birth, age, nationality, subject, gender and so on. In order to determine the student John from the other students, it is necessary to add additional logical constraints that contain a number of specifications capable of defining the target student, in order to restrict the list of other students. These log-

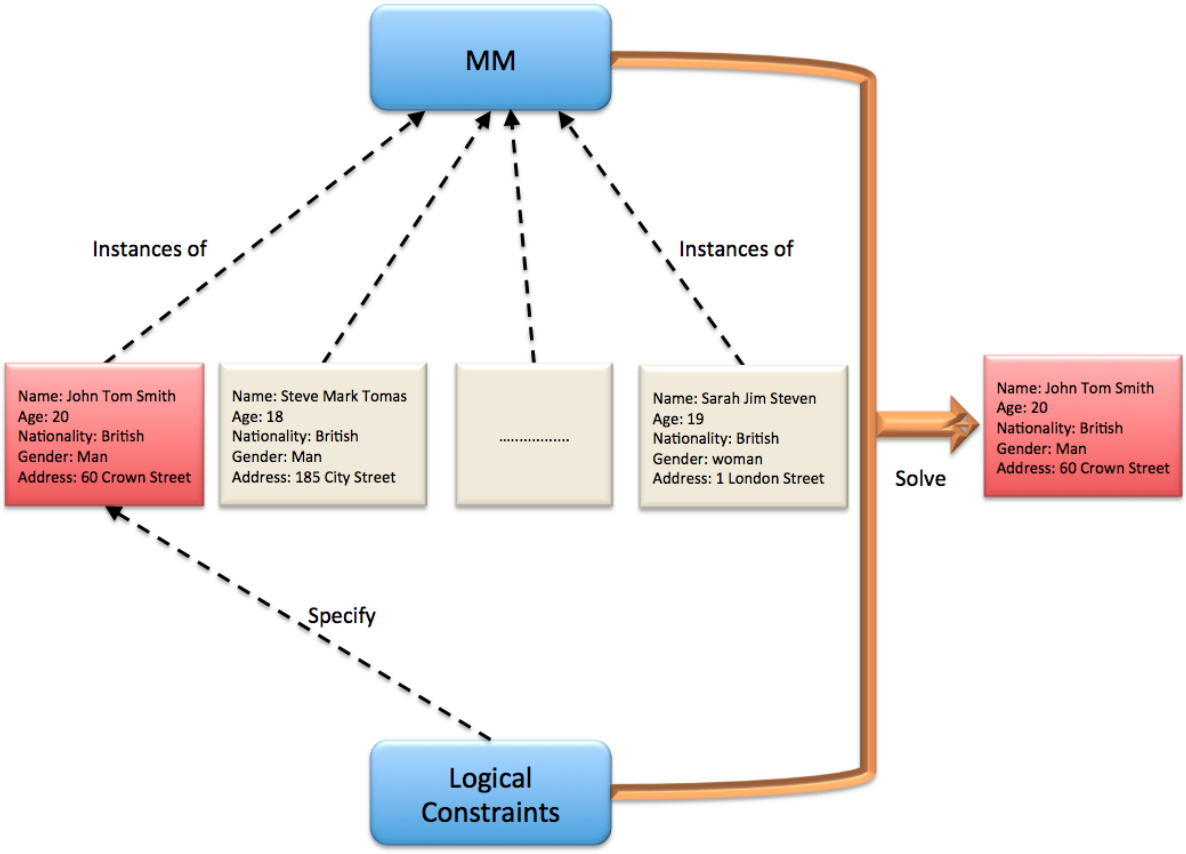


Figure 3.1: EMR mechanisms

ical constraints contain a list of student properties, including: $Student.FirstName = John$ &$Student.FatherName = Tom$ & $Student.LastName = Smith$ & $Student.Age = 20$ & $Student.Nationality = British$ & $Student.Gender = Man$, & $Student.Address = 60$ $Crown\ Street$. Specifying these properties of the student John, the solution of solving such logical constraints via constraint solver, produce only one solution, (i.e. the student John), that satisfies the logical constraints. In fact, the EMR technique is designed to be applicable to the automated instantiation of static and behaviour models. The following sections illustrate this technique in detail.

3.3 Application of EMR to Static Models

Software models can be categorised into two types: static models and dynamic models. As noted in Chapter 2, section 2.2, static models frequently focus on the structural aspects of the

system, i.e. relationships between packages, and demonstrating instance specifications (or relationships) between classes. The current section focuses on the representation of static models via EMR.

The EMR technique can be employed for a variety of static models, including the class diagram, object diagrams and the Entity Relation Diagram (ERD). In EMR, the diagram can be fully described by a set of logical constraints written in the context of the metamodel. This is illustrated in the example in Figures (3.2 and 3.3), adopted from [67]. The example depicts a home-automated system called 'Smart Home'. There are wide range of electronic and electrical devices to be found in the majority of homes, including lights, air conditioning systems, smoke detectors and televisions. Smart Home connects these devices and enables the home owner to monitor and control them using a software application. The home network also allows devices to coordinate their behaviour in order to fulfil complex tasks without human intervention. Meanwhile, sensors consist of devices capable of measuring the physical values of their environment, making them available to Smart Home.

Controllers activate devices which have a state that can be monitored and changed. All home devices are part of the Smart Home network, with their status being changed, either manually by residents, or from the Smart Home application.

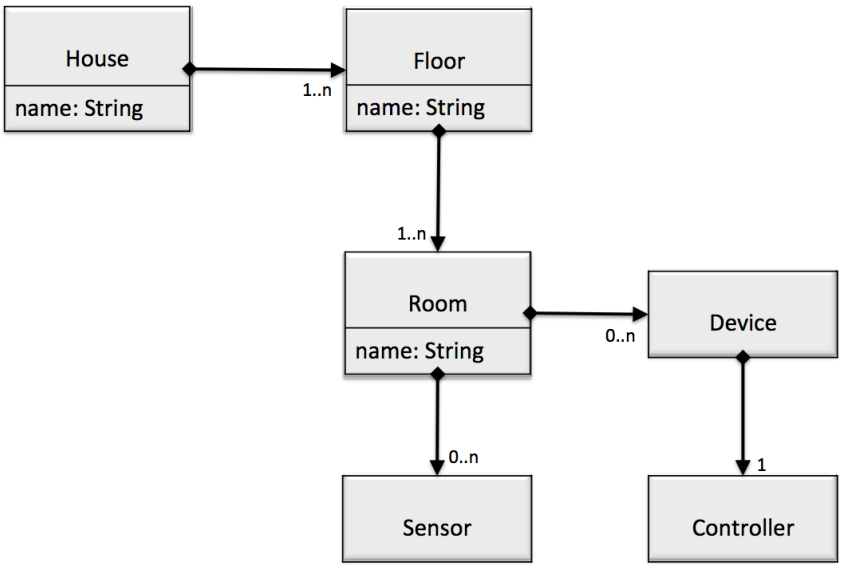


Figure 3.2: Smart Home MetaModel

Figure 3.2 demonstrates a simplified metamodel of Smart Home. A house contains a number of floors, each of which contains rooms. Each room contains sensors, and devices are also installed, each controlled by a controller. Figure 3.3 demonstrates an instance of the Smart Home metamodel, i.e. a concrete home automation system. The example house has only one floor, which contains only one bedroom, with one light sensor and two lights, each controlled by a light controller.

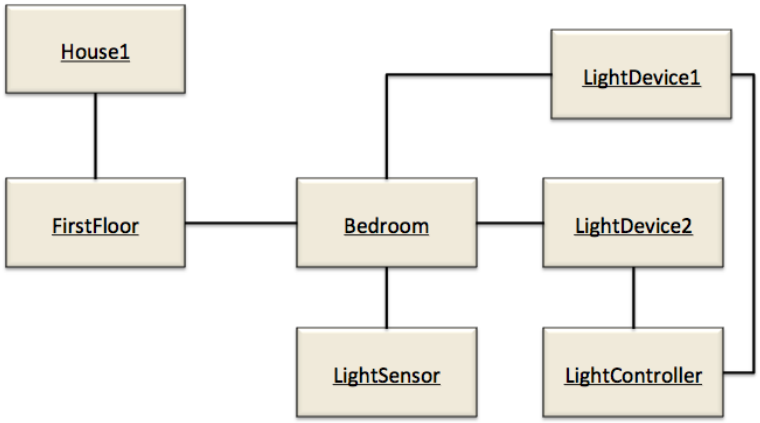


Figure 3.3: Smart Home Model

The metamodel in Figure 3.2 has many instances. Some houses contain two, or more, floors, and one or more bedrooms, etc. To restrict these instances, a set of logical constraints have been written on the Smart Home metamodel (Figure 3.2), which uniquely identifies the house model in Figure 3.3. The logical statements define each element by means of a diagram and its associations. The constraints also preserve the association multiplicity types (i.e. one-to-one relationships and one-to-many relationships), as demonstrated by the following code.

```
1   abstract sig House{floor: some Floor}
2   abstract sig Floor {Bedroom: some Room}
3   abstract sig Room {sen: set Sensors, Light: some Devices}
4   abstract sig Devices{ ControlledBy : one Controller}
5   abstract sig Sensors{}
6   abstract sig Controller{}
7   sig simpleHouse extends House{}
8   sig FirstFloor extends Floor{}
9   sig Bedroom extends Room{}
10  sig LightDevice extends Devices{}
11  sig LightSensor extends Sensors {}
12  one sig LightController extends Controller{}
13  fact { simpleHouse.floor in FirstFloor}
14  fact { Floor.Bedroom in Bedroom}
15  fact { Bedroom.sen in LightSensor}
16  fact { Bedroom.Light in LightDevice}
17  fact { LightDevice.ControlledBy in LightController}
18  run{} for 3 but exactly 1 House, exactly 1 FirstFloor , exactly 2 LightDevice, exactly 1
    Sensors, exactly 1 Bedroom
```

The code above is written in Alloy language. The Abstract signatures above (lines 1-6) define the metamodel elements (House, Floor, Bedroom, etc.). The associations between these elements are defined as a relationships (floor, bedroom, etc.). It should be noted that the reason of writing names in the relation is because Alloy relations must hold names. The keywords (e.g. $some, set, one$) demonstrate the representation of association multiplicity, as explained in section 2.8.1. For example, the association between House and Floor is one to many, and is defined in Alloy as $some$ keyword. In Alloy, $some$ means a cardinality of one or more instances of Floor. Lines 7-12 declare the model elements in which the sets of facts (lines 13-17) link the model element.

The run command in line 18 restricts the instances of the model enforcing constraints on the solver to produce only one solution, which contains only one house, one floor, one bedroom, two LightDevices and exactly 1 Sensor. It should be noted that there is no need to specify the controller, due to its definition as a unique set (i.e. one keyword before the signature). This restriction removes the further instances unrelated to the model, and which simply produce the

solution, uniquely identifying the model in Figure 3.4 (see below).

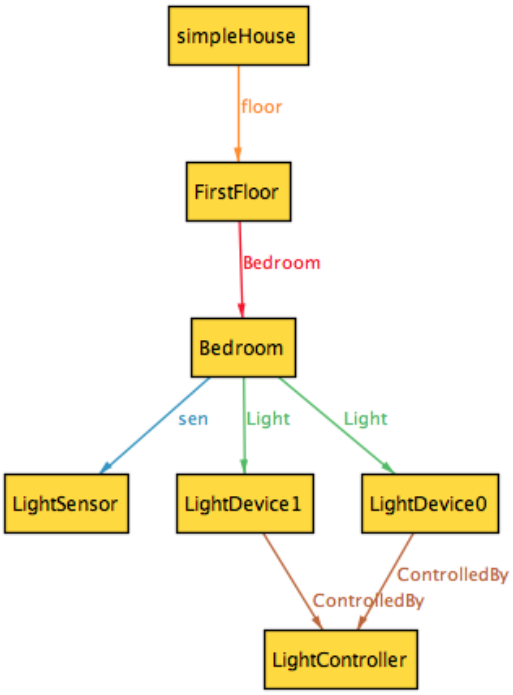


Figure 3.4: Alloy instance

3.4 Composition of Static Models

The composition of models is a process of combining two, or more, to create a single coherent model based on composition glue. The composition glue in this approach consists of a number of logical constraints. These specify which elements need to be composed, along with where the elements should be inserted, and the ways in which the composition process works to obtain the expected result.

The composition process in this technique is primarily aimed at generating a new model, consisting of all logical constraints associated with the original models in need of composition, along with additional constraints that describe the composition glue, as shown in Figure 3.5. The constraint solver then solves the new model and produces all possible solutions if all logical constraints of the input models are satisfied. Figure 3.5 depicts the mechanism of the composition, with M_1 and M_2 representing two input diagrams. Through *EMR*, two sets of con-

straints are produced, i.e. $\mathcal{L}_1$ and $\mathcal{L}_2$, in the metamodel of M_1 and M_2, each of which is uniquely identified with the original model. In order to compose the two models, a new model ($\mathcal{L}_3$) is generated, consisting of $\mathcal{L}_1$ and $\mathcal{L}_2$ and the glue $\mathcal{L}_g$. The glue ($\mathcal{L}_g$) matches the properties of the common elements of the two models, i.e. the elements names (name-based matching), and composes them. Moreover, the properties represented by model elements, such as attributes, are also matched and matched attributes will appear in the merged model only once. Finally, properties without any correspondence in the other model will be added to the composed model.

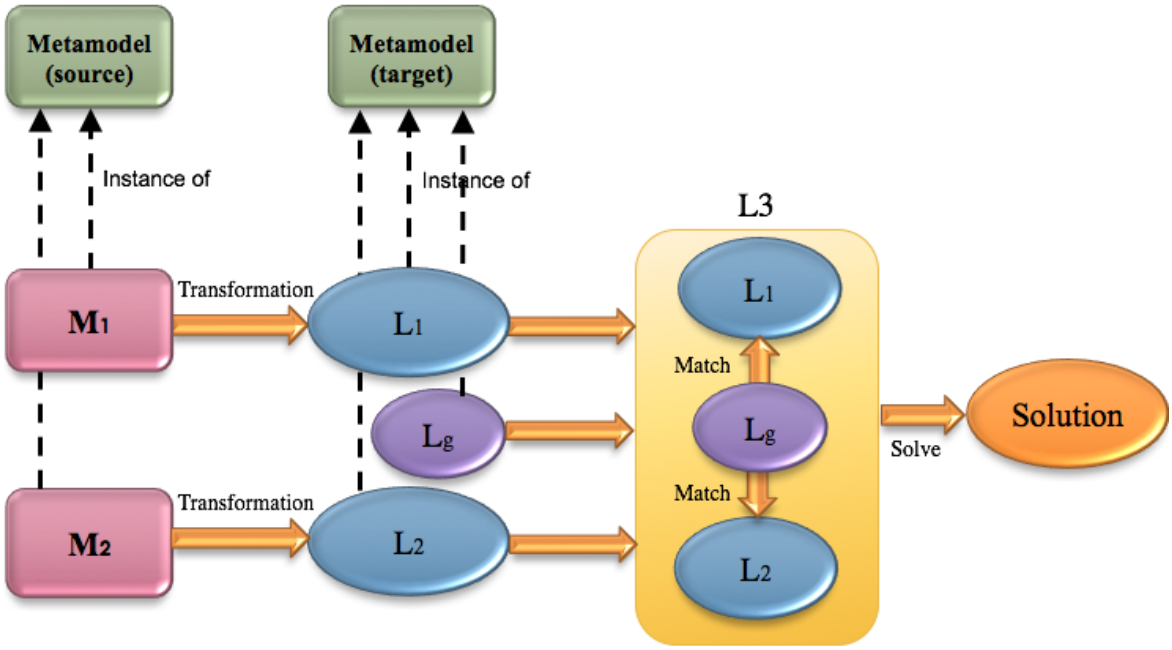


Figure 3.5: Model composition

The sets of constraints, $\mathcal{L}_1$, $\mathcal{L}_2$ and $\mathcal{L}_g$, then can be composed automatically, using a constraint solver (i.e. such as Alloy, Z3), producing a solution containing all elements of the input models, as well as preserving the associations between the elements. Further illustration about the generation of the composition glue will be presented in Chapters 4 and 6.

3.5 The Challenges of Behavioural Models

In the systems design, dynamic models focus on the behaviour of the system, which consists of observable information, exchanged between components within a system. Dynamic models

are frequently employed in software design to achieve a common understanding of the overall interactions within the system. Behaviour models frequently consist of two types: (1) abstract syntax and (2) dynamic representation (semantics). Abstract syntax is given by a metamodel, that defines all elements of a behaviour model and its possible relationships, which, in turn, describe the structural information underlying a design model. The dynamic representation (semantics) describes the behaviour of the system. The central concept of semantics is a trace of execution. "A trace of execution is a sequence of event occurrences ordered by time that corresponds to a system run" [116]. Thus, it describes information concerning a list of message exchanges corresponding to a system run.

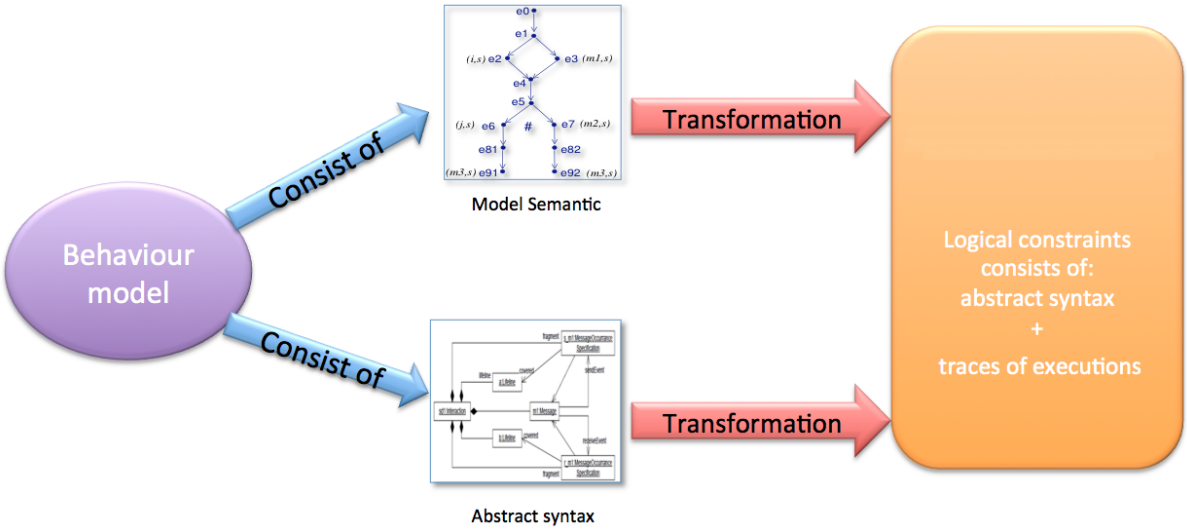


Figure 3.6: Behaviour Models

The semantics of behaviour models are not given in the metamodel, and must be defined separately. Currently, there is increasing acceptance that formal methods form an essential aspect of the design of any reliable complex software system [105]. This is due to formal methods having the potential to illustrate ambiguities and design faults, and thus avoid associated system failures. A number of possible semantics have been defined to describe the semantics of behaviour models, including: Labelled Event Structure (LES); Petri Net, Automata; and Labelled Transition System (LTS). In this current thesis, the dynamic interpretation of interactions, and their composition, is performed employing a Labelled Event Structure (LES), which, due to its simplicity, is suitable for describing semantics (i.e. traces of execution), and is capable of

directly capturing the available notions, e.g. sequential, parallel and iterative behaviour.

The representation of behaviour models is a challenging process, as it must take into account both the abstract syntax (static representation) and traces of execution (dynamic representation). Therefore, EMR needs to be enhanced by adding semantics, in order to obtain a solution containing the traces of the execution of the model. This is obtained by adding sets of logical constraints that capture traces of the execution of the model, i.e. if the traces represent sequential order, then the transformation adds a set of axioms that force the order of the events to be sequential. On the other hand, if the order is parallel, or a choice, then the constraint solver may produce more than one solution, each of which contains a trace in a different event order for the case of the parallel, or each solution contains a different choice of events for the case of choice. Therefore, the constraint solver generates all possible instances represented in the running of the original model. Although the EMR technique is compatible with different behaviour models containing traces of execution, the scope of this research focuses on the representation and composition of sequence diagrams.

3.5.1 Sequence Diagrams via EMR

As mentioned in previous sections, the research scope of this thesis is focused on sequence diagrams. Sequence diagrams are described in UML's superstructure specification [116], both through a concrete, and an abstract, syntax. As noted previously, the semantics of the sequence diagram are performed via LES. Due to the abstract syntax describing the structural information underlying a design model, the logical constraints by which it is represented can be generated employing the technique discussed in the static model via EMR. However, as noted in the previous section, the semantics differ, as they need to incorporate additional dynamic information, obtained from the LES interpretation to represent the behaviour of the model. Therefore, the logical constraints obtained by the Exact Metamodel Restrictions (EMR) have been extended to consider the dynamic (i.e. LES-based) interpretation.

The extended constraints define the sets of events in traces of execution of the model. Additionally, a number of axioms are written, enforcing the solver to generate a solution capturing

the behaviour of traces, including the sequential, alternative, and parallel order. The combination of the logical constraints that represent the abstract syntax, and the traces of execution, uniquely identify the original sequence diagram. This means that, starting from any UML sequence diagram and using a constraint solver for the sequence diagram metamodel and correct set of constraints, the constraint solver can be used to automatically recreate the original sequence diagram, i.e. when the constraint solver solves logical constraints; it generates the exact solution corresponding to the intended sequence diagram. Chapters 4 and 5 will illustrate in greater depth the process of generating the logical constraints of the sequence diagram via EMR.

3.5.2 Composition of Sequence Diagrams

Section 3.4 formed a discussion of the composition of static models. This section places additional focus on the composition of sequence diagrams. The composition of sequence diagrams focuses on composing the elements of the models (e.g. messages, lifelines, CombinedFragments, etc.) and the traces of execution (i.e. events and their relations). To do so, the composition of sequence diagrams might require some more options for the composition in addition to the syntactic matching, which gives the designer a way to influence the obtained composition by specifying behaviour that should never occur or sequences of events that must occur in a given order. In other words, it allows the designer to prioritise on specified behaviour. These options are called *behavioural composition glue*. Therefore, the interpretation of glue here is nonetheless more generic and not only a syntactic matching between component elements. The behavioural glue gives us a new set of constraints L_g which specifies how the models should be glued together to produce the intended composition. The guidance of such composition explained formally in the following section while the generation of the logical constraints will be demonstrate in the following chapter.

3.5.3 Composition Semantics

This section illustrates the composition semantics that have been used in this approach. In section 3.4, we explained the static composition. Static composition requires only the composition criteria, such as matching and composing elements with the same name. However, behaviour composition is different from static composition, which requires semantics to guide how the trace of execution of different models may be matched. This semantics describe formally the composition semantics and the composition glue in the context of LES.

In this semantics, we restrict ourselves to the composition of two diagrams. The case for the composition of a finite number of diagrams can be generalised from here. In the sequel, let SD_1 and SD_2 be two sequence diagrams, with sets of instances and messages given by I_1, I_2, Mes_1 and Mes_2 respectively.

When composing diagrams SD_1 and SD_2 we consider *interleaving* and *shared behaviour*. In the case of interleaving, the diagrams evolve completely autonomously of one another. That is, the *interleaving* of diagrams SD_1 and SD_2 is written $SD_1 \parallel SD_2$ and equivalent to $par(SD_1, SD_2)$. In other words, the composition is behaviourally equivalent to a diagram with a par fragment and two operands where each operand contains the behaviour described in SD_1 and SD_2 respectively.

The model for $SD_1 \parallel SD_2$, $M_{SD_1 \parallel SD_2} = (E, \mu)$, is an event structure where $Ev = Ev_1 \cup Ev_2$, all relations are preserved, and $\mu(e)$ is defined for all e iff $\mu_i(e)$ is defined for some $i \in \{1, 2\}$ in which case $\mu(e) = \mu_i(e)$. For shared instances $o \in I_1 \cap I_2$ we further match the initial events for o in Ev_1 and Ev_2. Recall that an *initial event* for an object is an event for which $\downarrow e = \{e\}$ which means that the local configuration only contains itself. We use $\downarrow Ev_o$ to indicate the singleton containing the initial event of instance o.

The composition of diagrams with *shared behaviour* is written $SD_1 \parallel_G SD_2$ where G indicates the *composition glue*.

We define the composition of two models formally in two stages. First we define the model obtained by syntactic matching of objects and messages of both models. We then take the

glue constraints and apply a restriction on the matched composed model that satisfies the glue constraints.

Let $\Delta \subseteq L_1 \times L_2 \cup I_1 \times I_2$ be a binary relation over labels or instances satisfying if $(l, l') \in \Delta$ and $(l, l'') \in \Delta$ then $l' = l''$; and if $(l', l) \in \Delta$ and $(l'', l) \in \Delta$ then $l' = l''$. We call Δ a *matching* over labels and instances. Let $\overline{Ev_1}$ (and similarly $\overline{Ev_2}$) correspond to the set of events in Ev_1 with a label not matched in Δ.

Definition 7. *Let $M_1 = (E_1, \mu_1)$ and $M_2 = (E_2, \mu_2)$ be models for sequence diagrams SD_1 and SD_2, and Δ be a matching over labels and instances. $SD_1 \parallel_\Delta SD_2$ is a* matched composition model *for Δ given by $M_\Delta = (E, \mu)$ such that events in M_Δ are given by*

$$Ev = \overline{Ev_1} \cup \overline{Ev_2} \cup$$
$$\{(e_1, e_2) | (L(e_1), L(e_2)) \in \Delta\} \cup$$
$$\{(e_1, e_2) | (e_1 \in\downarrow Ev_{i_1}, e_2 \in\downarrow Ev_{i_2} \text{ and } (i_1, i_2) \in \Delta)\}$$

The labels are unchanged, that is, $\mu(e) = \mu_i(e)$ for $e \in \overline{Ev_i}$ with $i \in \{1, 2\}$ and $\mu(e_1, e_2) = \mu_1(e_1) = \mu_2(e_2)$. Event relations in M_Δ are derived from the relations in M_1 and M_2 as follows $(e_1, e_2) \rightarrow^* e$ iff $(e_1 \rightarrow_1^* e$ or $e_2 \rightarrow_2^* e)$; $e_i \rightarrow e_i'$ iff $e_i \rightarrow_i^* e_i'$; and $(e_1, e_2) \rightarrow^* (e_1', e_2')$ iff $(e_1 \rightarrow_1^* e_1'$ and $e_2 \rightarrow_2^* e_2')$. Similarly for the conflict relation with additional conflict derived from propagation over causality.

According to the above definition, the event pairs (e_1, e_2) in Ev correspond to events matched by Δ or denoting initial events for shared objects. Relations and labels are preserved in the composition as expected.

If the model obtained above is a valid labelled event structure then a composition for SD_1 and SD_2 according to Δ exists. Otherwise the models are not composable.

Proposition 1. *Let $M_1 = (E_1, \mu_1)$ and $M_2 = (E_2, \mu_2)$ be models for sequence diagrams SD_1 and SD_2, and Δ be a matching over instances and labels. The diagrams are* composable *according to Δ iff the matched composition model $M_\Delta = (E, \mu)$ is a well defined labelled event structure.*

A case that illustrates a non composable model is one where the same two messages (say m_1 and m_2) are sent in the reverse order in two diagrams. The model obtained by matching the respective send/receive events in both diagrams would lead to an invalid labelled event structure as the model would contain a cycle which is not allowed. We illustrate the idea of shared behaviour further with the example from Figure 2.5 to obtain the composition of `sd1` and `sd2` .

We consider the matching of messages and lifelines with the same name, i.e., messages `m1` and `m2`, and lifelines for object `a` and object `b`. There is a matched composition model M_Δ for `sd1` and `sd2` as shown in Fig. 3.7.

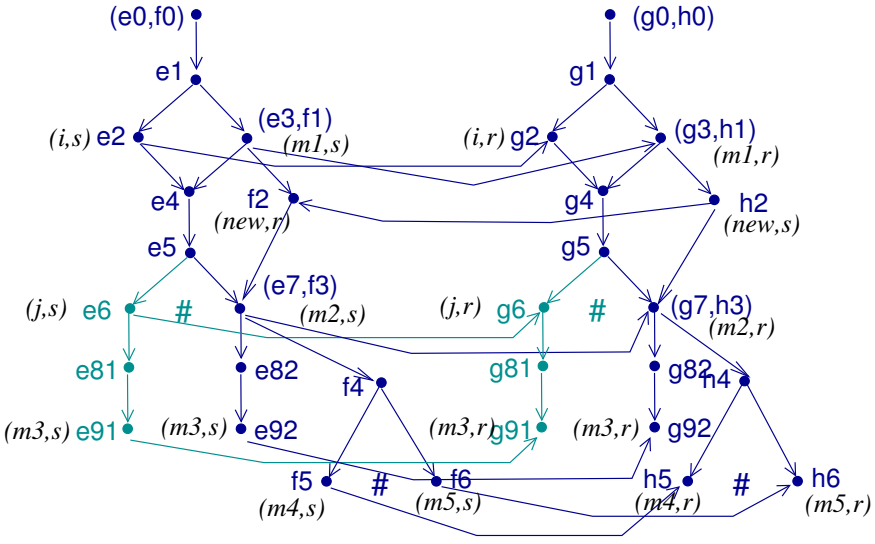


Figure 3.7: Matched composition model

It shows the matched initial events (e.g., (e_0, f_0)) and events matched by Δ (e.g., (e_3, f_1) for label (m_1, s)). Event relations are derived from the original relations and any conflict that arises from propagation over the extended causality relation. In this case, $e_6\#(e_7, f_3)$ since $e_6\#e_7$ and consequently also $e_6\#f_4$, and so on.

We want to allow a designer to add further constraints on the expected composition by for example specifying behaviour that should never occur (forbidden events) or sequences of events that must occur in a given order, and so on. This can be seen as a way to give priority to certain specified interactions, and eliminates some of the possible traces in the composed model.

In the following, let $M_1 = (E_1, \mu_1)$ and $M_2 = (E_2, \mu_2)$ be composable models over Δ for

sequence diagrams SD_1 and SD_2 with Δ a matching over labels and instances. Let $M_\Delta = (E, \mu)$ be the matched composed model obtained, and Γ be the set of maximal configurations (traces) in M_Δ.

Definition 8. *A* behavioural glue *for $M_\Delta = (E, \mu)$ is given by $G = (Ev_g, \rightarrow_g^*, \#_g, Fv_g)$ where $Ev_g, Fv_g \subseteq Ev$ are subsets of events that occur in E, and $\rightarrow_g^*, \#_g \subseteq Ev_g \times Ev_g$ are binary relations (causality and conflict) defined over the events in Ev_g. Events in Fv are forbidden events.*

A behavioural glue G as defined above may contain relations over events which disagree with the relations in M_Δ. However, we can always obtain an equivalent glue G' that preserves the relations in $M_\Delta = (E, \mu)$ by considering all the events that violate the original relations as forbidden events.

Definition 9. *A composed model $SD_1 \|_G SD_2$ for relation preserving glue G is given by $M_G = (E_G, \mu_G)$ such that it corresponds to M_Δ by removing all traces $t \in \Gamma$ such that $Fv \cap t \neq \emptyset$.*

Consider the two cases of behavioural glue as shown in Fig. 3.8.

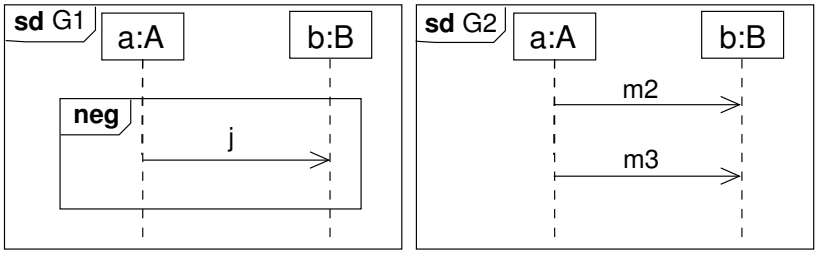


Figure 3.8: Examples of behavioural glue

The behavioural glue `G1` imposes that the occurrence of message `j` is forbidden in the composed model. Glue `G2` imposes that for `m3` to occur, `m2` must have happened before.
For `G1` we have $G_1 = (\emptyset, \emptyset, \emptyset, \{e_6, g_6\})$ where the events associated to message j are forbidden. This means that the composed model for `sd1` and `sd2` wrt `G1` removes all traces which contain events e_6 and g_6 from the matched composition model shown in Fig. 3.7. Since the events in $\downarrow e_5$ (and similarly $\downarrow g_5$) belong to another valid trace they are not removed. We obtain a composed model which is identical to the matched composition model but where the highlighted relations and events have been removed (i.e., events $e_6, e_{81}, e_{91}, g_6, g_{81}, g_{91}$ and relations).

For G2 we consider an equivalent glue which preserves the relations, namely $G_2 = (Ev_{g2}, \rightarrow^*_{g2}, \emptyset, Fv_{g2})$ where $Ev_{g2} = \{(e_7, f_3), (g_7, h_3), e_{92}, g_{92}\}$, $Fv_{g2} = \{e_{91}, g_{91}\}$ and the causality relation is such that $\rightarrow^*_{g2} = \{((e_7, f_3), e_{92}), ((g_7, h_3), g_{92})\}$. In this case we need to remove all traces which contain e_{91} and g_{91} from the matched composition model shown in Figure 3.7. The composed model for sd1 and sd2 wrt G2 coincides with the composed model wrt G1 described earlier. This follows because the traces affected by the forbidden events are the same.

3.6 Chapter Summary

This chapter has outlined a technique for the representation and composing of static and behavioural models at metamodel level, known as EMR. The outline of the method involves the creation of logical constraints that uniquely identify the model. To combine the models, logical constraints that glue the two models were produced. Some of these logical constraints declare matching elements, while others are used to enforce behaviour involved in the composition, e.g. specifying behaviour that should never occur, or sequences of events that must occur in a given order. This makes it possible for a designer to give priority to certain specified interactions, which is considered in the solution by eliminating unwanted traces from an initial matched model obtained. In order to ensure the correctness of the composition process, the semantics of the composition have been formalised with the assistance of LES. Chapter 4 demonstrates the first implementation of EMR to generate logical constraints that uniquely identify sequence diagrams and compose them via Alloy.

CHAPTER 4

COMPOSITION OF SEQUENCE DIAGRAMS VIA ALLOY

4.1 Overview

This chapter uses the EMR technique discussed in Chapter 3 to transform and compose sequence diagrams via Alloy. The chapter consists of two main sections; in section 4.2, the transformation rules are presented, which transform the UML sequence diagram elements into Alloy. These rules create sets of logical constraints through EMR, which uniquely characterise each diagram by restricting the metamodels.

In section 4.3, the static and behavioural glue for combining the models are described. These types of glue feature constraints indicating how elements from the input models can be matched.

The transformation and composition process between sequence diagrams and Alloy is challenging, as creating logical constraints for large sequence diagrams is time-consuming and prone to human error. As a result, this work utilises a Model-Driven Architecture (MDA) approach to automating the transformation between the sequence diagram and Alloy.

4.2 Transformation of Sequence Diagrams to Alloy

As indicated in the overview, this approach is automated; making use of MDA techniques to transform sequence diagrams into Alloy. The model transformation process is hereby described

in three stages:

- Mapping the metamodels of the source to target models.
- Establishing transformation rules to map the elements of the sequence diagrams and Alloy.
- Implementing the transformation rules (which will be discussed in Appendix A).

In order to use an MDA methodology to automate the transformation between the sequence diagrams and Alloy, metamodels for the source and target models need to be constructed, specifying the elements of the sequence diagrams that will be mapped to Alloy. The metamodels of the sequence diagram and Alloy are presented in section 2.3.1.2, Figure 2.6 and section 2.5.1, Figure 2.10. In this approach, the complete features of the Alloy language are not considered, but instead only those features that are used in the present transformation are depicted. In the next section, the respective transformation rules for mapping the elements of the sequence diagram metamodel to the elements of the Alloy metamodel are described in greater depth.

4.2.1 Transformation Rules

This section describes the model transformation process, whereby any sequence diagram models conforming to the sequence diagram metamodel in Figure 2.6 are transformed into Alloy. This requires a set of seven transformation rules to be defined for mapping the elements of the sequence diagrams into Alloy features. These transformation rules are written in Java. Figure 4.1 presents an overview of the correspondence between the elements in the sequence diagrams and Alloy.

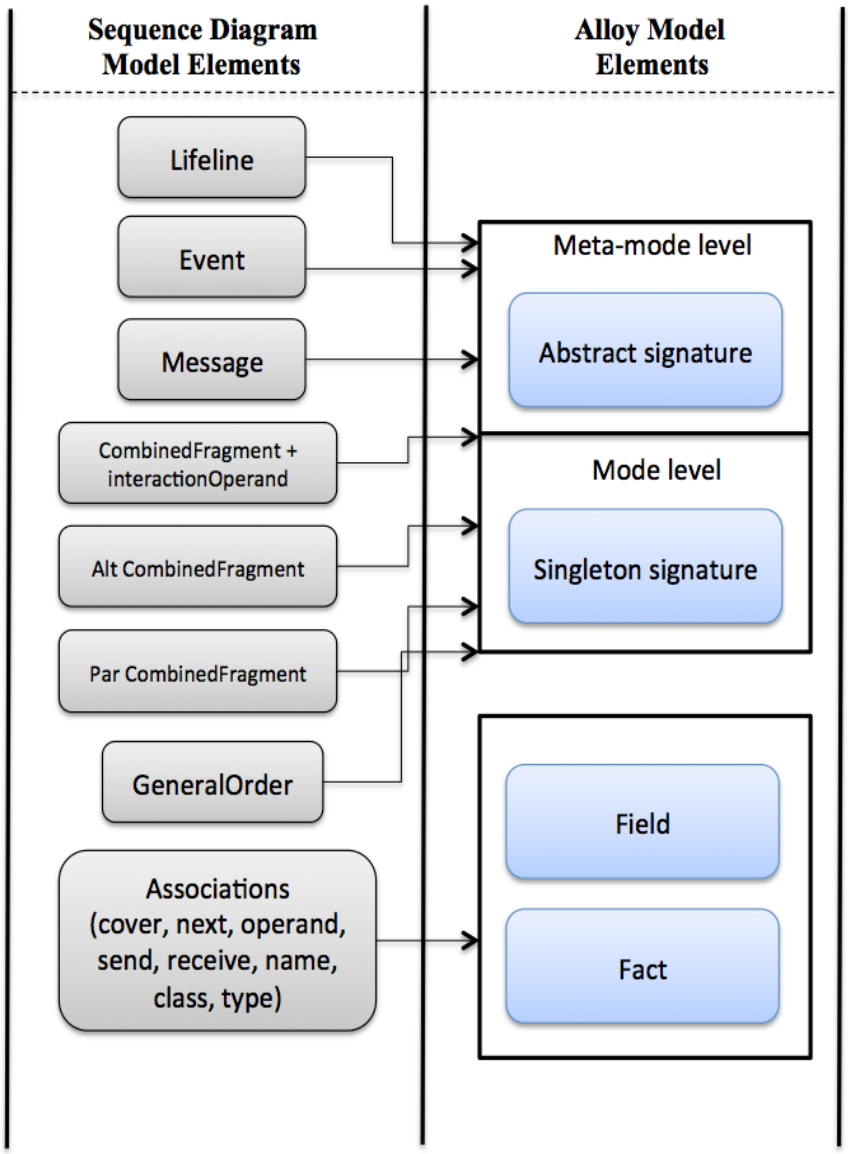


Figure 4.1: Overview of the thesis approach

Figure 4.1 gives an overview of the transformation rules to be presented in this section. Each element of the sequence diagrams maps to its correspondent in Alloy. In total, this chapter proposes seven transformation rules and these represent the main rules that consider both the structure and dynamic interpretation of a sequence diagram when producing an Alloy model [1].

The model is obtained via EMR; that is, by considering the abstract syntax of a diagram and the constraints obtained from the dynamic (LES-based) interpretation, the exact solution in Alloy, corresponding to the intended sequence diagram, is generated. Moreover, our approach

[1] In Alloy, the associations are not defined separately, but within the rules.

is such that if an Alloy model can be solved, it generates all possible solutions each of which corresponds to a run of the original sequence diagram and in accordance to the formal semantics. The syntax and semantics of Alloy are apparent in the following rules and code snippets, but certain key notions will first be introduced.

Data domains for sequence diagrams are defined using signatures, given by the keyword *sig* and represented as sets. Just as in object-oriented languages, a signature may extend another signature, in which case the domain defined by the first is a subset of the domain of the extended signature. A signature that is declared independently of any other is called a 'top-level signature'. Extensions of a signature are mutually disjoint, as are top-level signatures. A signature can also be *abstract*, in which case its domain will only contain elements belonging to its extending signatures. As shown in Figure 4.1, for each element in the metamodel of the sequence diagram, an abstract signature is generated and for each element at model level, a *singleton* signature is also generated.

In Alloy, signatures may contain *fields*, which are captured by relations. Each relation must be given a name. Axioms in Alloy are called *facts* and can also be given a name. *Fields* and *facts* will be used in this approach to capture the association between the elements, as depicted in Figure 4.1. Moreover, *fields* are used to define the association name, e.g. *next*, *cover*, etc., whereas facts are used to enforce the restrictions of the association.

Next, the transformation rules are described to demonstrate challenging aspects of the transformation. In the transformation rules, the mapping between the elements of the metamodel and model element is illustrated. In addition, we illustrate the transformation of the sequence diagram, sd1 in Figure 2.5.

4.2.2 Rule 1- Transforming Lifelines

As previously established, for each element in the metamodel, the transformation generates an abstract signature. Therefore, the transformation maps a lifeline element in the metamodel of the sequence diagram into an *abstract* signature in Alloy. This *abstract* signature, called a *lifeline*, as shown in Figure 4.2.

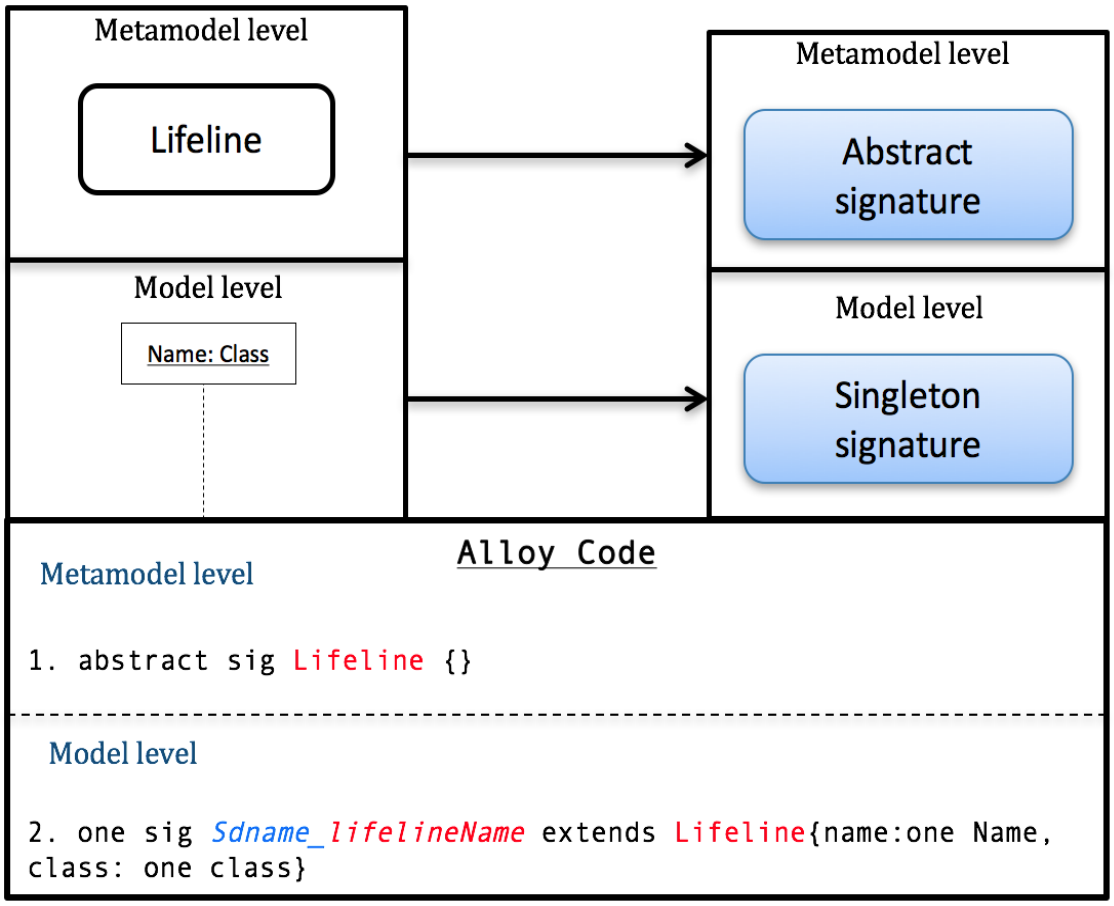


Figure 4.2: Lifelines transformation rule

This means that the *abstract signature* lifeline has no elements, except those that belong to its extension. Moreover, for each concrete lifeline in a sequence diagram, a one-line declaration is obtained, seen in Figure 4.2 (line 2). The multiplicity keyword *one* in the declaration indicates that there is precisely one instance of the signature. This means that the solver will only produce one instance for each lifeline signature, which will uniquely identify the original lifeline in the sequence diagram.

A lifeline in the sequence diagram has a *name* and belongs to a *class*. Thus, each lifeline signature in Alloy has two fields: *name* and *class*, which define the *name* and *class* of the lifeline. For example, the lifelines in sd1 (Figure 2.5), as described in section 2.1.1, consist of two lifelines and will be transformed into the following Alloy code:

```
abstract sig Lifeline {}
one sig sd1_a extends Lifeline{name: one a, class: one A}
one sig sd1_b extends Lifeline{name: one b, class: one B}
```

Remark: The name of the signature at model level in all transformation rules consists of two parts, as shown in Figure 4.3.

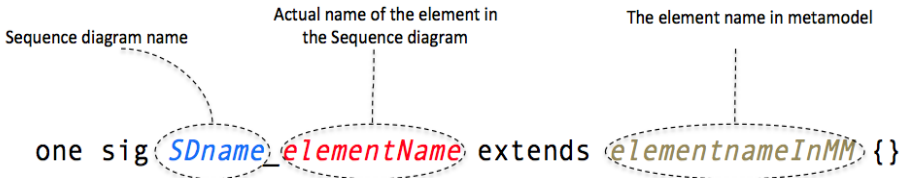


Figure 4.3: Naming in an Alloy signature

The first part of the signature name indicates the name of the sequence diagram the element belongs to, e.g. sd1_a, whereas the second part indicates the actual name of the element in the sequence diagram, e.g. sd1_a. The reason for adding the name of a sequence diagram is that in Alloy, two signatures cannot exist with the same name. However, the name may be repeated across different sequence diagrams. Therefore, problems can be avoided by adding information about which diagram it belongs to, as shown in Figure 4.3.

4.2.3 Rule 2- Transforming Events

Event in this approach represents the class *OccurrenceSpecification* in the metamodel. *OccurrenceSpecification* in turn refers to a moment in time (an event) at the beginning or end of a message [116]. Each event in the sequence diagram appears on precisely one lifeline, whereas a lifeline can have one or more events (as shown in the metamodel in Figure 2.6). Moreover, the events on each lifeline must be ordered from top to bottom.

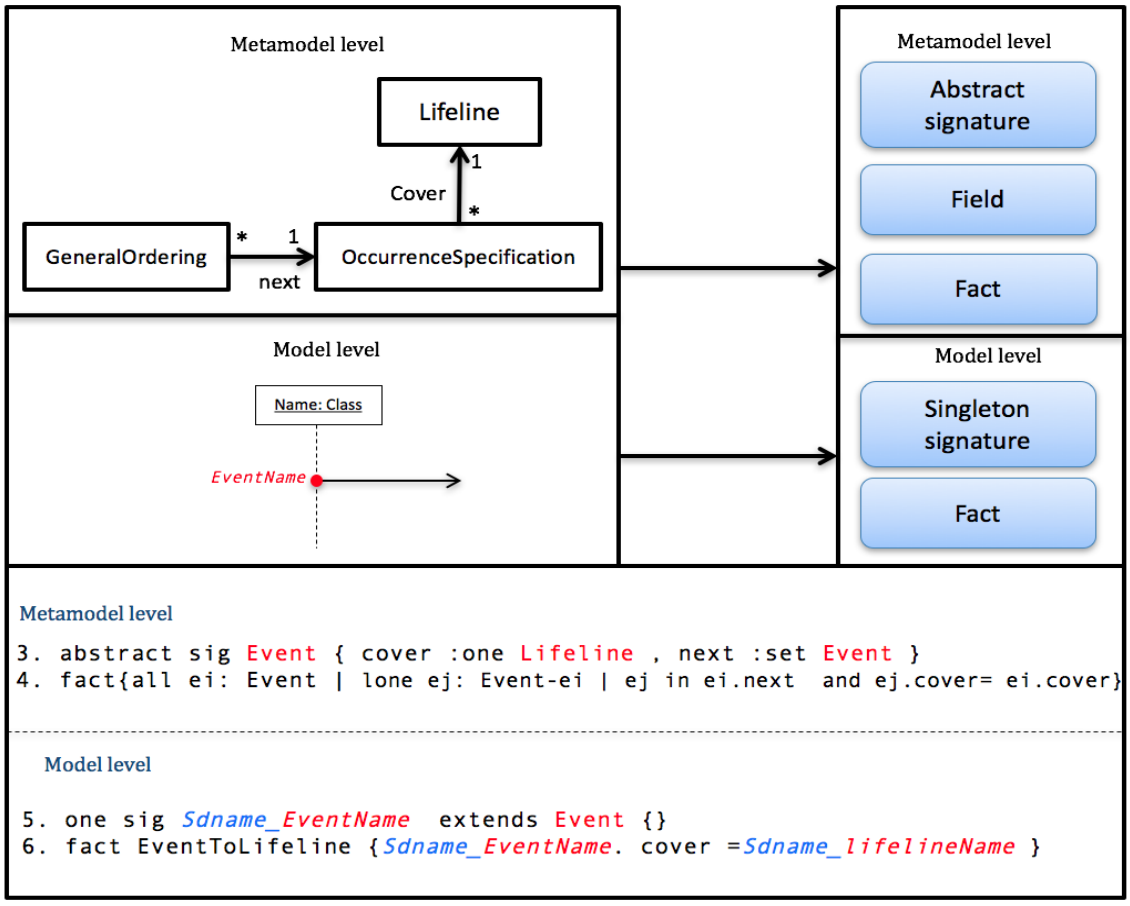


Figure 4.4: Events transformation rule

The rule, as shown above in Figure 4.4, creates the domain 'Event' at metamodel level. In addition, the abstract signature has two fields: *cover* and *next*. The field, *cover* corresponds to a relationship with the lifeline it belongs to. The metamodel shows that the *OccurrenceSpecification* (event) can appear on precisely one lifeline. Hence, this association multiplicity has been written in Alloy as the keyword, *one* at the beginning of the field, which means the events can be linked to just one lifeline(line 3).

Further to the above, the field, *next* (line 3) corresponds to a relationship with a set of events. The keyword, *set* in a field states that a single event can link to *zero* or more events in the diagram. However, this relation is restricted by the fact at metamodel level (see Figure 4.4, line 4). The *fact* states that the event can have at most one *next* linked to it in the same lifeline, specifying the multiplicity restrictions between the *GeneralOrder* and *OccurrenceSpecification* in the metamodel. This corresponds exactly to the definition of the *OccurrenceSpecification* and

its association multiplicity with a lifeline in the UML specification [116]. The *GeneralOrder* is explained in more detail below in rule 7.

Next, similar to the other element at model level, a one-line declaration is obtained for each event from amongst the sequence diagram events in line 5 (see Figure 4.4). Finally, a fact *EventToLifeline* in line 6, is generated to associate the model events to the lifelines.

For example, sd1 (Figure 2.5) consists of 10 events and is transformed into the following Alloy code:

```
one sig sd1_e2 extends Event {}
one sig sd1_e3 extends Event {}
lone sig sd1_e6 extends Event {}
lone sig sd1_e7 extends Event {}
one sig sd1_e9 extends Event {}
one sig sd1_g2 extends Event {}
one sig sd1_g3 extends Event {}
lone sig sd1_g6 extends Event {}
lone sig sd1_g7 extends Event {}
one sig sd1_g9 extends Event {}
```

The above code declares the sd1 events mapped from the LES in Figure 2.9. Notice that this minimises of what was shown in the previous section, with LES. In our semantics, there are events to indicate the beginning and end of an interaction fragment, as well as communication events. In Alloy, we omitted the fragment events to reduce the size of the model. Thus, the events declared above correspond to the messages *send* and *receive*.

For consistency, the same event names are used here as are used in the semantic model for the same diagram (see Figure 2.9). Incidentally, it is not necessary to duplicate events '$e9$' or '$g9$', because Alloy will produce two solutions to represent two possible alternative executions. The following fact, *EventToLifeline* connects the model events to the lifelines.

```
fact EventToLifeline {
    e2.cover = sd1_a and g2.cover = sd1_b and e3.cover = sd1_a and g3.cover = sd1_b and e6.
        cover = sd1_a and g6.cover = sd1_b and e7.cover = sd1_a and g7.cover = sd1_b and e9.
        cover = sd1_a and g9.cover = sd1_b }
```

4.2.4 Rule 3- Transforming Messages

As previously established, a message represents a communication object shown as an arrow that connects the respective lifelines [116]. In the metamodel, a Message has two MessageEnds, namely a SendEvent and a ReceiveEvent, which cover a Lifeline. A ReceiveEvent must always be preceded by a SendEvent.

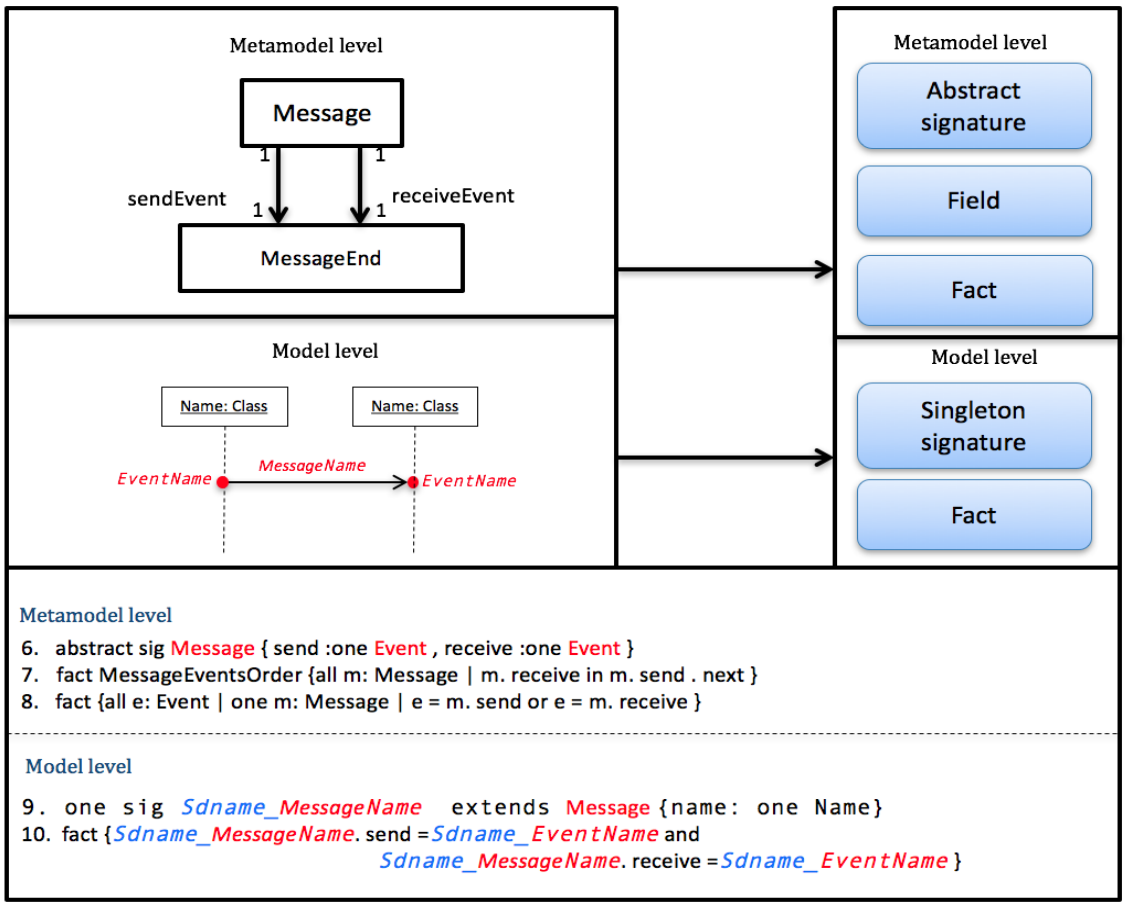


Figure 4.5: The messages transformation rule

As Figure 4.5 shows, the transformation rule maps the message element in the metamodel to an *abstract* signature and *fact*. The abstract signature declares the message domain, as shown in line 6. This signature has two fields, *'send'* and *'receive'*; both corresponding to one event [1]. The facts on lines 7-8 describe two constraints over the elements in the domain, as captured in the rule. The first fact (line 7), called *MessageEventsOrder*, states that for any message,

[1] In this approach, it is assumed that the message always has *'send'* and *'receive'* events

m, *m.receive* must belong to the set: *m.send.next*. This means that m.receive must always be preceded by *m.send*. The second fact (line 8) states that all events considered are either *'send'* or *'receive'* events. The constraints (lines 7, 8) must be satisfied in all sequence diagram messages, in order to ensure the correctness of the transformation. In other words, these constraints are designed to make sure that the Alloy model always produces a correct message, based on the definition of the sequence diagram messages in [116].

Line 9 declares the concrete message. As shown, the message contains a field name, which defines the actual message name in the diagram. Finally, the fact in line 10 connects the message with its send/receive events. The following snippet of code defines the sd1 messages:

```
one sig sd1_i extends Message { name :one i}
one sig sd1_m1 extends Message { name :one m1}
lone sig sd1_m2 extends Message { name : one m2}
lone sig sd1_j extends Message { name : one j}
one sig sd1_m3 extends Message { name :one m3}
```

Some of the messages in the Alloy code above are declared as *lone*; a multiplicity keyword in Alloy meaning 0 or 1, while others are declared as *one*, meaning exactly one. This relates to the fact that messages within an alternative CombinedFragment are not guaranteed to occur. This will be explained in more detail later, in an alternative CombinedFragment rule. In order to associate messages and events, a fact is added to specify this as the code below shows.

```
fact { sd1_i.send =e2 and sd1_i.receive =g2 and
       sd1_m1.send =e3 and sd1_m1.receive =g3 and
       sd1_j.send =e6 and sd1_j.receive =g6 and
       sd1_m2.send =e7 and sd1_m2.receive =g7 and
       sd1_m3.send =e9 and sd1_m3.receive =g9}
```

4.2.5 Rule 4- Transforming CombinedFragment

According to the UML specifications [116], a CombinedFragment has an InteractionOperator, given by type, and one or more InteractionOperands. An InteractionOperand covers a set of Events, CombinedFragments, or both.

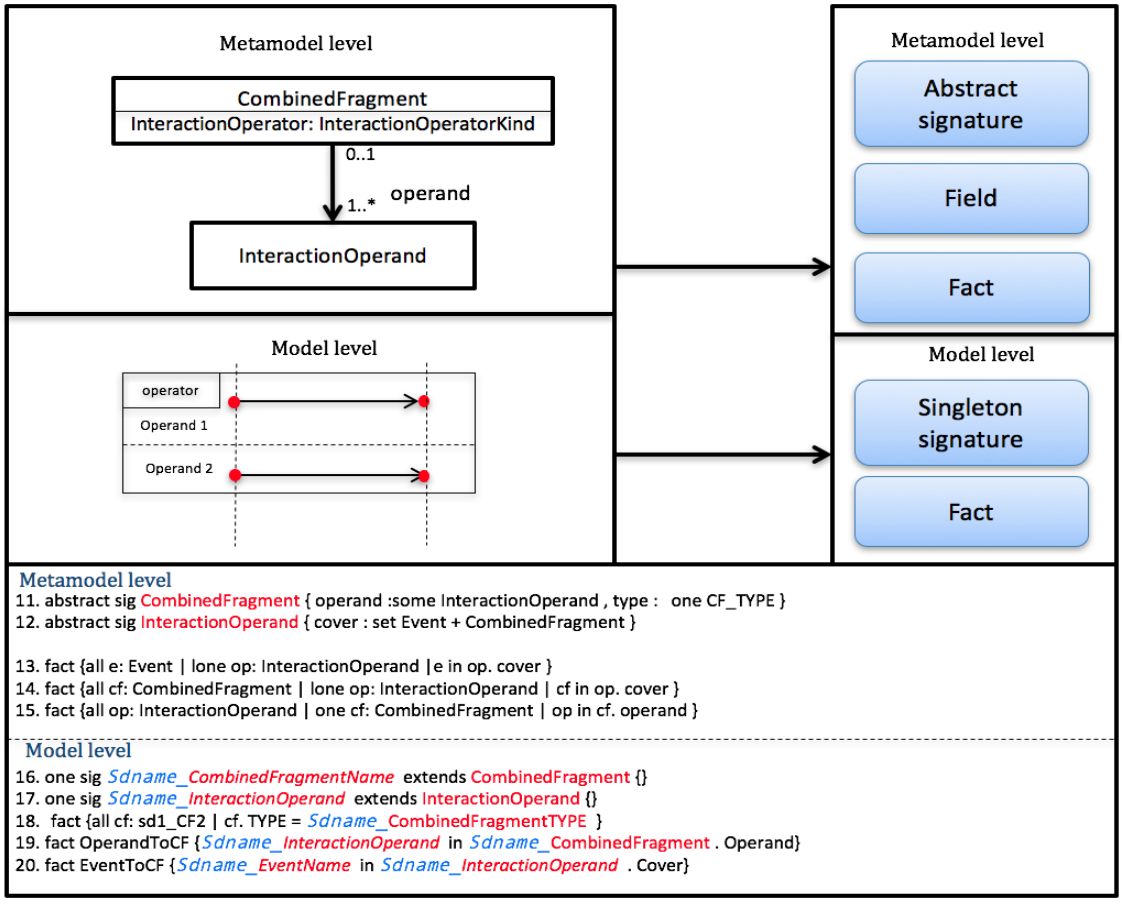


Figure 4.6: The CombinedFragment transformation rule

Lines 11-12 in Figure 4.6 define the metamodel elements, which map the CombinedFragments and InteractionOperands to abstract signatures. The abstract signatures for CombinedFragments consist of two fields: *operand* and *type*. The *operand* field shows that the CombinedFragment contains one or more InteractionOperand, whereas the *type* field specifies the InteractionOperator, such as *par* or *alt*. The abstract signatures for the InteractionOperand contain a field called *cover*, which shows that the InteractionOperand covers a set of events, CombinedFragments, or both. In addition, three facts impose further constraints on the elements of these domains; the fact on line 13 states that every event belongs to at most, one InteractionOperand and the fact in line 14 states that every CombinedFragment belongs to, at most, one InteractionOperand, indicating fragment nesting. The fact in line 15 then states that all InteractionOperands are operands to at most, one CombinedFragment. Lines 16-17 define the CombinedFragments and their InteractionOperands at modelling level. The fact in line 18 is

used to specify the type of CombinedFragment operator. The *OperandToCF* fact connects each InteractionOperand to its CombinedFragment, while the fact, *EventToCF* connects the events belonging to the CombinedFragment to the corresponding InteractionOperands. Thus, the constraints defined above uniquely identify the CombinedFragment of the sequence diagram.

4.2.6 Rule 5- Transforming Alternative CombinedFragment

As stated in the Background Chapter, section 2.2.1, the CombinedFragment, *alt* consists of two or more InteractionOperands. Each InteractionOperand describes a choice of behaviour. Only one of the alternative InteractionOperand is executed if the guard expression (where present) is evaluated as *'true'*.

```
// alt : exactly one operand will be executed
fact  Alt - Execution {all cf: CombinedFragment |  (cf. TYPE =    cf_TYPE_ALT ) => # cf.
    operand = 1}
```

In order to preserve the semantics of alternative CombinedFragments, the above fact states that exactly one InteractionOperand is executed. Note that # in fact, $AltExecution$ corresponds to the Alloy's cardinality operator. A consequence of this fact is that every time we run the code a different set of events (associated with a particular InteractionOperand) may be executed, but every time we only execute one InteractionOperand of an alternative CombinedFragment.

The Alloy code lines presented below describe an alternative CombinedFragment with two InteractionOperands and no guards, as is the case for the second CombinedFragment from *sd1*, shown in Figure 2.5.

```
one sig sd1_CF2 extends CombinedFragment {}
lone sig sd1_CF2_Op1 extends InteractionOperand {}
lone sig sd1_CF2_Op2 extends InteractionOperand {}
fact {all cf: sd1_CF2 | cf. TYPE = CF_TYPE_ALT }
```

The first three lines in the Alloy code above define the CombinedFragment and its InteractionOperands. The *lone* keyword may be noted at the beginning of the InteractionOperand signatures; this is necessary, as only one InteractionOperand will be able to execute in accordance

with the *Alt-Execution* fact. The fact in the last line specifies the type of CombinedFragment as an alternative. The following snippet of code shows two facts connecting the CombinedFragment with its InteractionOperands and the InteractionOperands with their events (declared in rule 2).

```
fact OperandToCF {
      sd1_CF2_Op1 in sd1_CF2.operand
      sd1_CF2_Op2 in sd1_CF2.operand }

fact EventToCF {
    e6 in sd1_CF2_Op1 . cover and g6 in sd1_CF2_Op1 . cover
    and e7 in sd1_CF2_Op2 . cover and g7 in sd1_CF2_Op2 . cover }
```

The fact *OperandToCF* connects each InteractionOperand of the second CombinedFragment of $sd1$ to its CombinedFragment, while the fact *EventToCF* connects the events declared in lines 15-17 which belong to this CombinedFragment to the corresponding InteractionOperands.

4.2.7 Rule 6- Transforming Parallel CombinedFragment

In Alloy, the representation of a parallel CombinedFragment is similar to that of an alternative CombinedFragment, but without the fact, *AltExecution*. The parallel CombinedFragment with two InteractionOperands is described by the snippets of Alloy code presented below, as is the case for the first CombinedFragment from $sd1$, shown in Figure 2.5.

```
one sig sd1_CF1 extends Combinedfragment{}
one sig sd1_CF1_Op1 extends Operand{}
one sig sd1_CF1_Op2 extends Operand{}
fact {all cf: sd1_CF2 | cf. TYPE = CF_TYPE_PAR }
fact{
      sd1_CF1_Op1 in CF.operand
      sd1_CF1_Op2 in CF.operand}
fact EventToOp{
     sd1_e2 in sd1_CF1_Op1.cover and sd1_g2 in sd1_CF1_Op1.cover
      sd1_e3 in sd1_CF1_Op2.cover and sd1_g3 in sd1_CF1_Op2.cover}
```

The transformation of a parallel CombinedFragment declares the InteractionOperands as *one* (the keyword *one* at the beginning of the signature), since all InteractionOperands must

occur at all times. The Alloy model containing a parallel CombinedFragment must show a parallel execution of *sd1_CF1_Op1* and *sd1_CF1_Op2*; in other words, the events covered by each InteractionOperand are not explicitly related by *next* and can thus occur in an arbitrary order. This is in accordance with the LES semantics presented earlier in Chapter 2, section 2.2.2. It implies a concurrency relationship between events in different InteractionOperands, whilst the events within an InteractionOperand remain ordered in the usual way by the *next* relation.

Finally, a rule is added to capture the notion of *GeneralOrdering* from the interaction meta-model, whereby a binary relationship is captured between two *OccurrenceSpecifications* events.

4.2.8 Rule 7- Transforming GeneralOrder

GeneralOrdering represents a binary relationship between two events. This is specified in Alloy by the logical constraint called *GeneralOrder*, which specifies the order in which all messages and their underlying events occur along the lifelines of the corresponding object instances. The transitive closure of the general ordering is irreflexive.

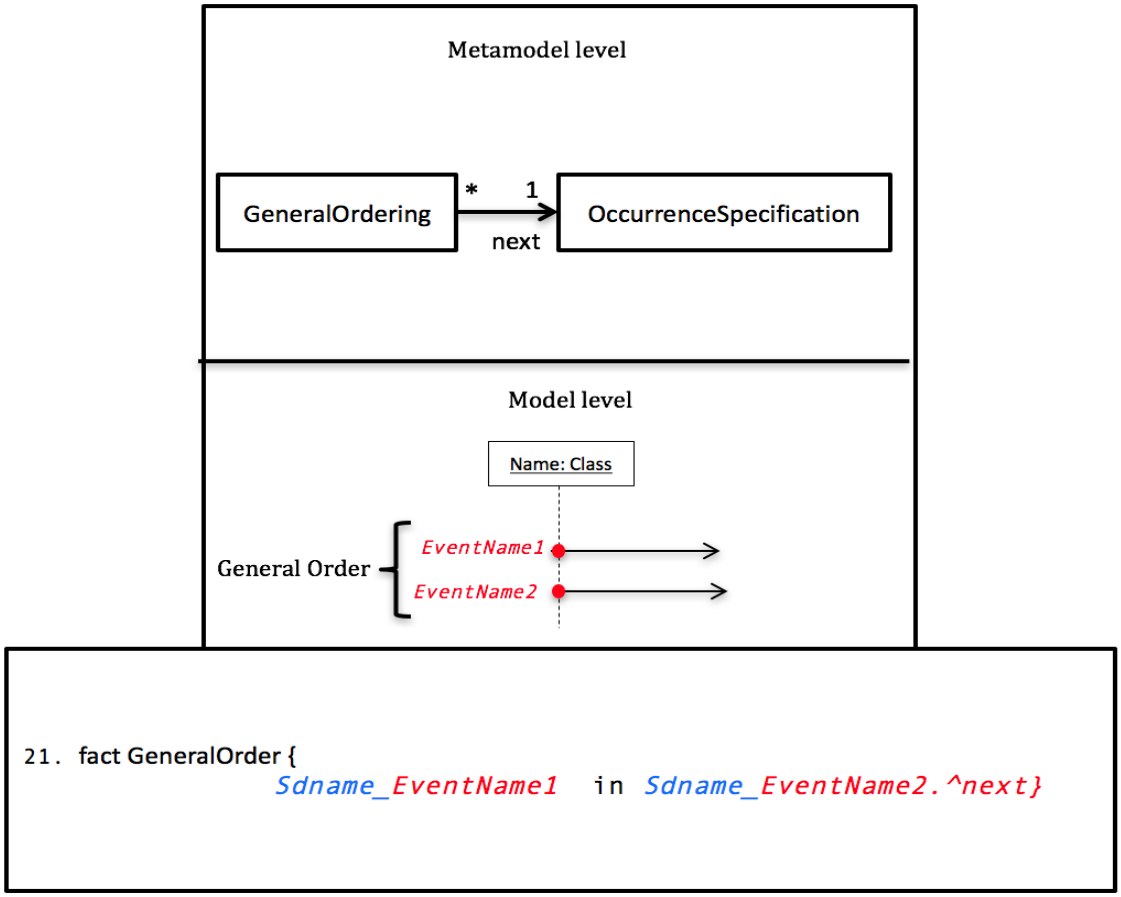


Figure 4.7: GeneralOrder in Alloy

In the case of a basic sequence diagram without CombinedFragments, this implies a total ordering along the events of the lifeline. It is specified in Alloy by another logical constraint called *GeneralOrder* (see Figure 4.7). This fact specifies the order of all events occur along the lifelines. In the fact (Figure 4.7-line 21), we make use of the unary operator ^c to denote the transitive closure of c. The following code depicts the order of the elements in the sd1 Figure. 2.5.

```
fact GeneralOrder {
  all l: sd1_a + sd1_b , ev1:sd1_CF1.operand.cover ,
     ev2:sd1_ CF2.operand.cover | ev1.cover = l
     and ev2.cover = l => ev2 in ev1.^ next
     and
  all l: sd1_a , ev1:sd1_CF2.operand.cover ,
     ev2:e9 | ev1.cover = l => ev2 in ev1.^ next
   and
```

```
all l: sd1_b , ev1:sd1_cf2.operand.cover ,
  ev2:g9 | ev1.cover = l => ev2 in ev1.^ next}
```

The fact in the Alloy code above states that all events *ev1* and *ev2* such that *ev1* belongs to the first CombinedFragment and *ev2* belongs to the second CombinedFragment, if they cover the same lifeline then *ev2* belongs to the transitive closure of *ev1.next*, that is, it necessarily occurs after *ev1*. Note that `ev1` $\neq$ `ev2` since they are elements from different extensions of *CombinedFragment* and necessarily disjoint in Alloy. The above code shows that the occurrence of an event *e9* or *g9* must be preceded by the occurrence of events covered by the second CombinedFragment. In other words, the sending/receiving of message *m3* can only occur if the CombinedFragments have executed beforehand.

4.3 Composition of Sequence Diagrams in Alloy

Model composition requires a *composition glue* to combine the partial models, as mentioned in Chapter 3. Using this approach, two kinds of composition will be demonstrated here: *syntactic glue* and *behavioural glue*. The mechanism of each will be explained in greater depth in the following subsections.

4.3.1 Syntactic Glue

In Alloy, the composition conditions mentioned in Chapter 2, section 2.3 could be encoded by adding *facts*, which must be satisfied to match and compose the overlapping elements. The goal of the syntactic glue is to match the syntactic properties, such as the *name* and *type* (if present) of the overlapping elements in the input diagrams and then to compose them.

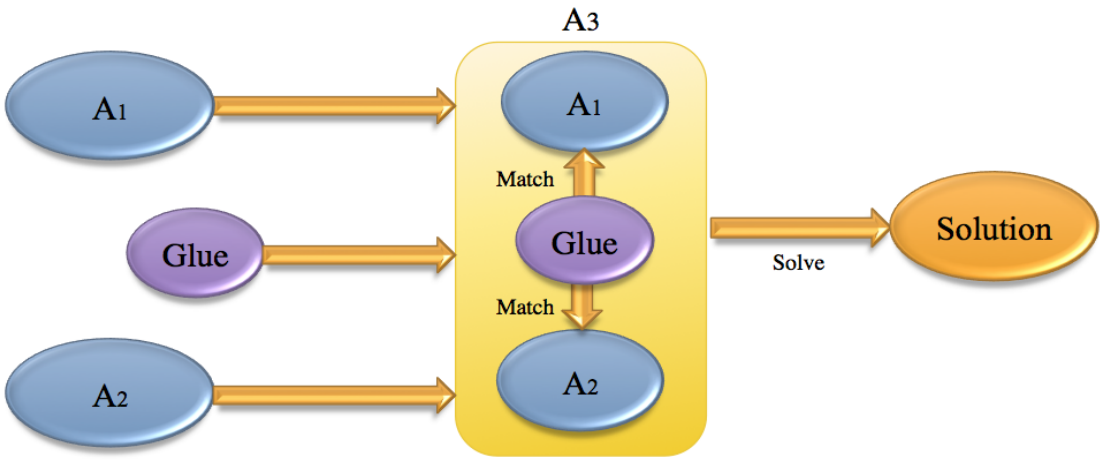


Figure 4.8: Composition mechanism in Alloy

The procedure for composition in Alloy (Figure 4.8) can be explained as follows. First, a new Alloy model, *A3*, is generated, representing the result of merging the original models. Second, all elements of *A1* are copied to *A3*. Third, all elements of *A2* are copied, except for duplicate elements, such as abstract signatures that are shared in the two models. Moreover, the abstract signatures represent the elements of the metamodel, such as the lifeline, message or event already defined in any sequence diagram. Therefore, Alloy does not permit the duplication of any abstract signatures that are already defined. Thus, only abstract signatures of the second Alloy model, *A2*, which are not included in the *A1* model, may be copied, such as the abstract signature of CombinedFragments and InteractionOperands, if the *A2* model contains CombinedFragments.

Fourth, all element signatures of *A2* that correspond to *A1* elements must be changed from *one* to *lone*, in order to be composed. Changing the signature to *lone* enables the atom, which represents the signature in the Alloy solution (instance) to be removed, by changing the cardinality of the signature to '0'. The elements of *A2* that do not have any correspondence will remain as *one*, in order to occur in the final composed instance. The following subsection will explain in greater depth the mechanism of composition for the main sequence diagram elements, such as lifelines, messages and CombinedFragments.

4.3.1.1 Composition of Lifelines:

For any two lifelines declared as matching in the glue, a matching fact is generated. This fact will match properties, such as the lifeline *names* and *classes* (types). For example, if there are two Alloy models, *A1* and *A2*, representing two sequence diagrams each with two lifelines and these lifelines have the same name and class, in order for them to be composed, the following fact must be specified.

```
fact lifelineEquality {
all L1_1: sd1_L1, L2_1: sd2_L1 |(L1_1.name = L2_1.name && L1_1.class = L2_1.class) =># L2_1
    =0}
all L2_1: sd1_L2, L2_2: sd2_L2 |(L2_1.name = L2_2.name && L2_1.class = L2_2.class) =># L2_2
    =0}
```

The fact, *lifelineEquality* defines that if the names and classes of the lifelines are matching, then the lifelines will be composed into `sd1_L1, sd1_L2`, while `sd2_L1, sd2_L2` of Alloy model *A2* will be removed. To illustrate this further, the mechanism of the facts *lifelineEquality* is first to match the lifeline names and classes, such that (`L1_1.name=L2_1.name`) and (`L1_1.class = L2_2.class`), If one property does not match, the *Unsat Core* will highlight the unmatched properties. Otherwise, the lifelines will be matched. Secondly, the atoms of the lifelines signatures, i.e. (`sd2_L1, sd2_L2`) of model *A2* will be removed in the *A3* solution by changing the signature cardinality to '0', i.e.(`#L2_1 =0` and `#L2_2=0`). Finally, all events linked to `sd2_L1` and `sd2_L2`, which are removed, will be linked to `sd1_L1` and `sd1_L2` as the following fact shows. This is due to `sd2_L1` and `sd2_L2` will not occur in the *A3* solution (instance). Therefore, the events must be linked to the composed lifeline, as shown in Figure 4.9.

```
fact EventToLifeline {
sd2_e1.cover = sd1_L1 and sd2_g1.cover = sd1_L2
.....}
```

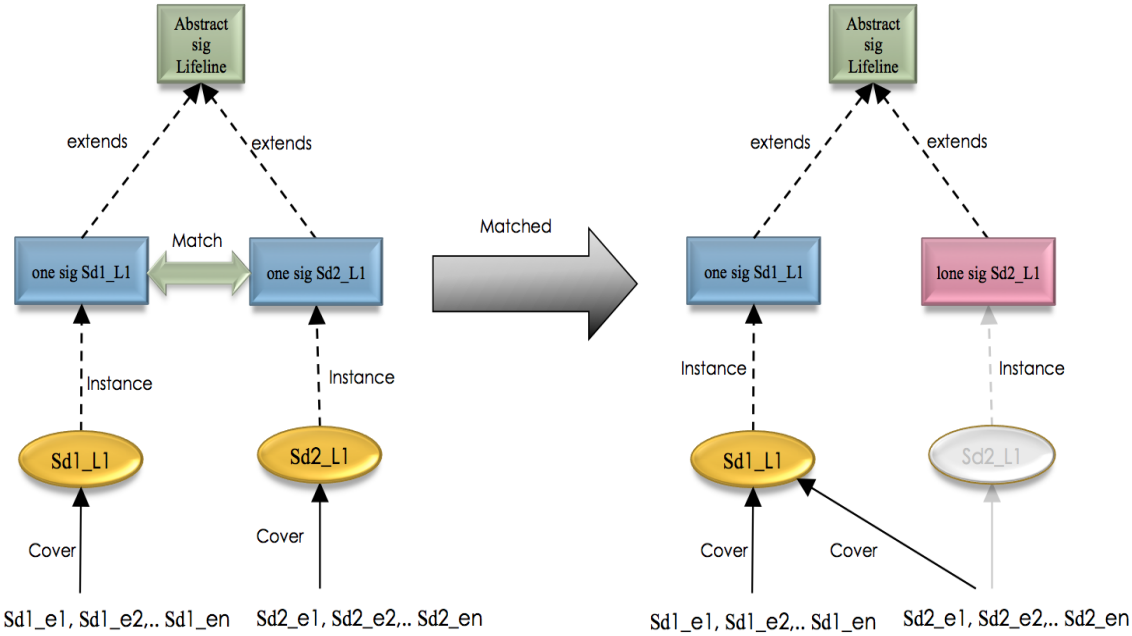


Figure 4.9: Lifeline compositions

For example, consider the diagrams, sd1 and sd2 (see Figure 2.5), each with two lifelines that have the same name and class. In order to compose these lifelines, the following fact must be specified:

```
fact lifelineEquality {
all L1: sd1_a , L2: sd2_a |
(L1.name=L2.name && L1.class=L2.class) =># L2 =0
all L3: sd1_b , L4: sd2_b |
(L3.name=L4.name && L3.class=L4.class) =># L4 =0}

fact EventToLifeline {
    sd2_e2. cover = sd1_a and sd2_g2. cover = sd1_b and sd2_e3. cover = sd1_a and sd2_g3.
        cover = sd1_b and sd2_e6. cover = sd1_a and sd2_g6. cover = sd1_b and sd2_e7. cover =
        sd1_a and sd2_g7. cover = sd1_b and sd2_e9. cover = sd1_a and sd2_g9. cover = sd1_b }
```

4.3.1.2 Composition of Messages:

The same composition procedure is applied for messages; all message signatures of *A2* that correspond to *A1* messages must be changed from *one* to *lone*, in order to be composed. However, the *A2* messages, which do not have any correspondence, will remain as *one*. Thus, for any

two common messages with the same *name* and '*send*' and '*receive*' from the same lifelines - meaning that the lifelines the messages send and receive from have the same *name* and *class* - a matching fact is generated. The fact will match the message name and lifelines these messages send and receive from. For example, consider two Alloy models *A1* and *A2*, representing two sequence diagrams in Figure 4.10.

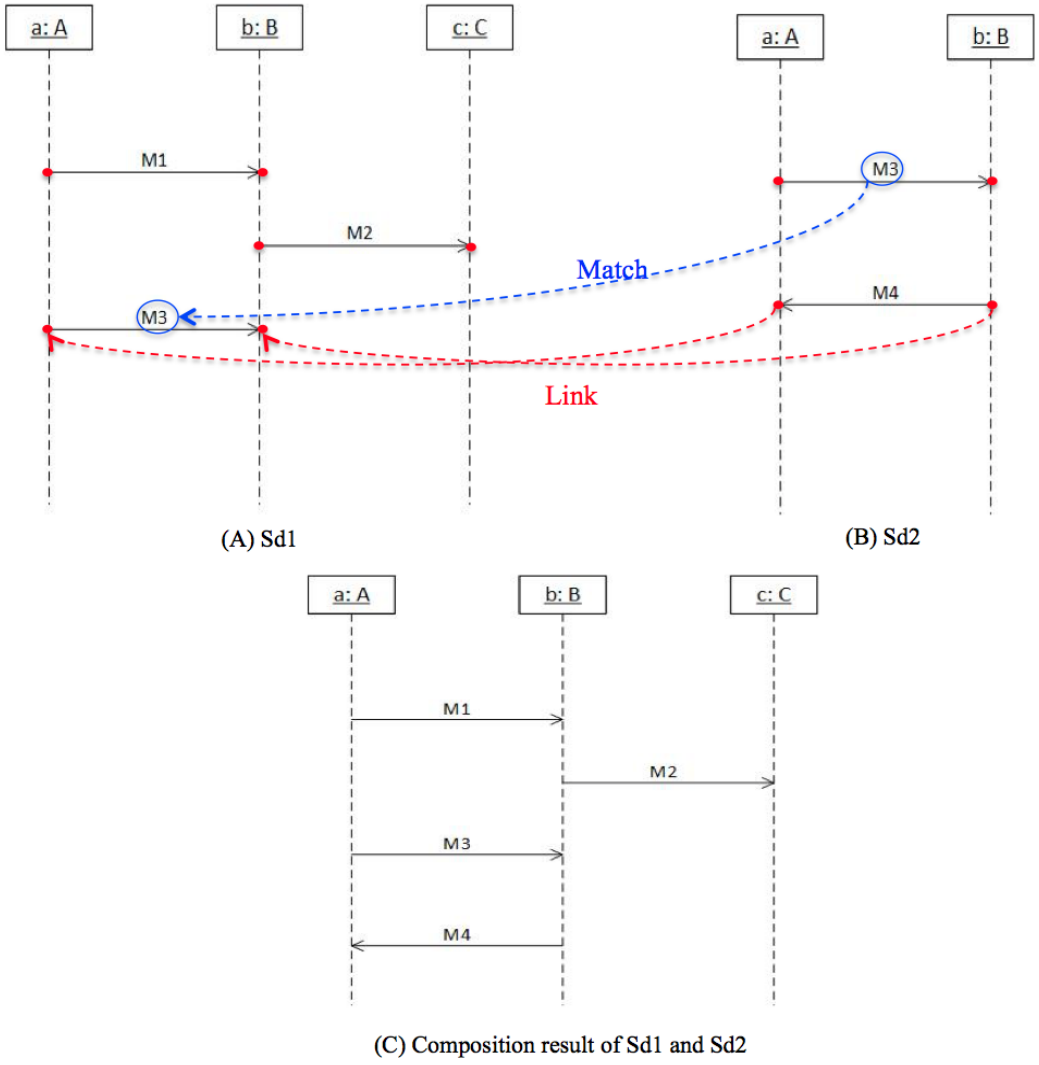


Figure 4.10: A composition example

Messages *M3* in each diagram are matched, due to these messages having the same name and their send and receive lifelines are matched. In order to compose messages with the same name from each of the models, the following fact must be specified:

```
fact {all m1: sd1_M2, m2: sd2_M2|(m1.name = m2.name && m1.send = m2.send && m1.receive =
    m2.receive)=> #m2 =0 && #m2.send =0 && #m2.receive =0}
```

Once the above fact returns *true*, then it will be composed into one; namely `sd1_M2`, while `sd2_M2` and its 'send' and 'receive' events will be removed; similar to what was explained for the lifeline composition. Finally, in relation to composing messages, the composed message events, 'send' and 'receive', as they are removed, are replaced with their equivalent message events to apply the behavioural environment of both models to this message (see Figure 4.11). Thus, all messages events linked to 'send' and 'receive' events of `sd2_M2` via the *next* relation will be linked to 'send' and 'receive' events, `sd1_M2`. This means that the events occurring before and after the 'send' and 'receive' events of `sd1_M2` will become before and after send and receive events, `sd1_M2`, due to `sd2_M2` being removed (Figure 4.11).

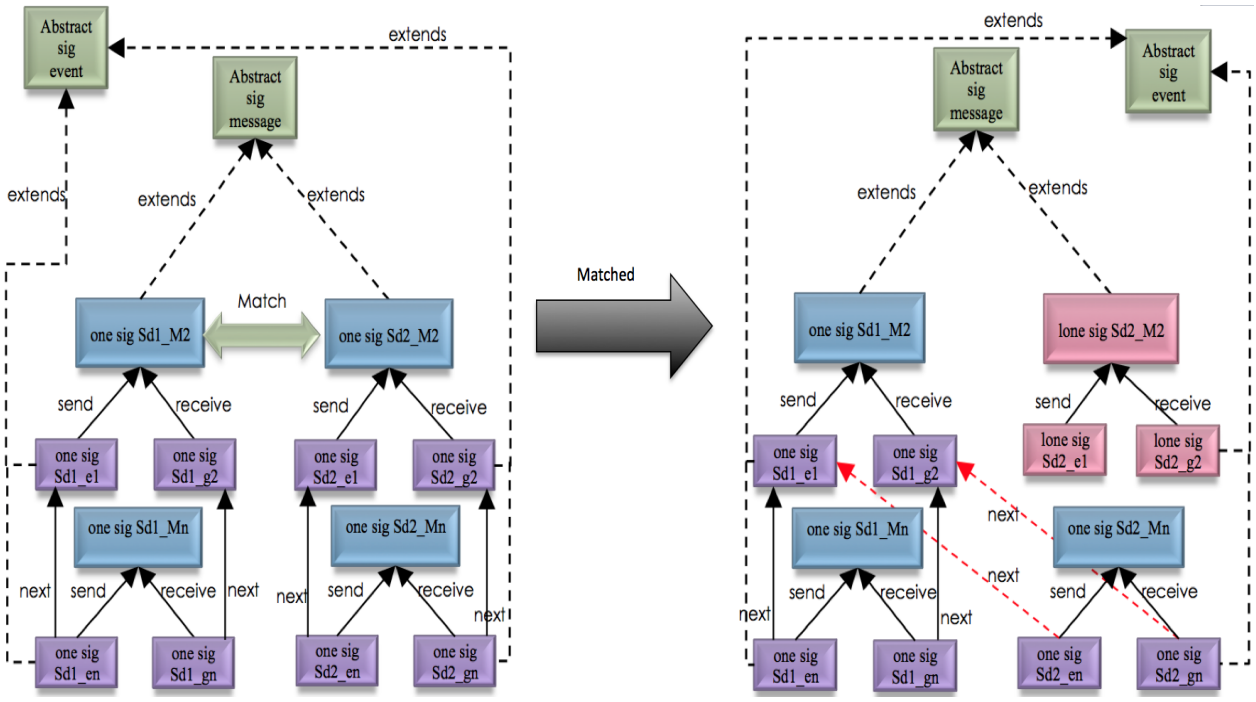


Figure 4.11: Messages compositions

4.3.1.3 Composition of a CombinedFragment:

In some cases, the sequence diagram is consists of one or more CombinedFragments. The composition of the sequence diagrams with a CombinedFragment is more complex, because it

presents complex behaviour, which requires extra logical constraints. In fact, there are many cases of composing sequence diagrams with one or more CombinedFragments. However, in this section, two such cases are described, whereas the other scenarios can be composed using the same techniques presented in this section.

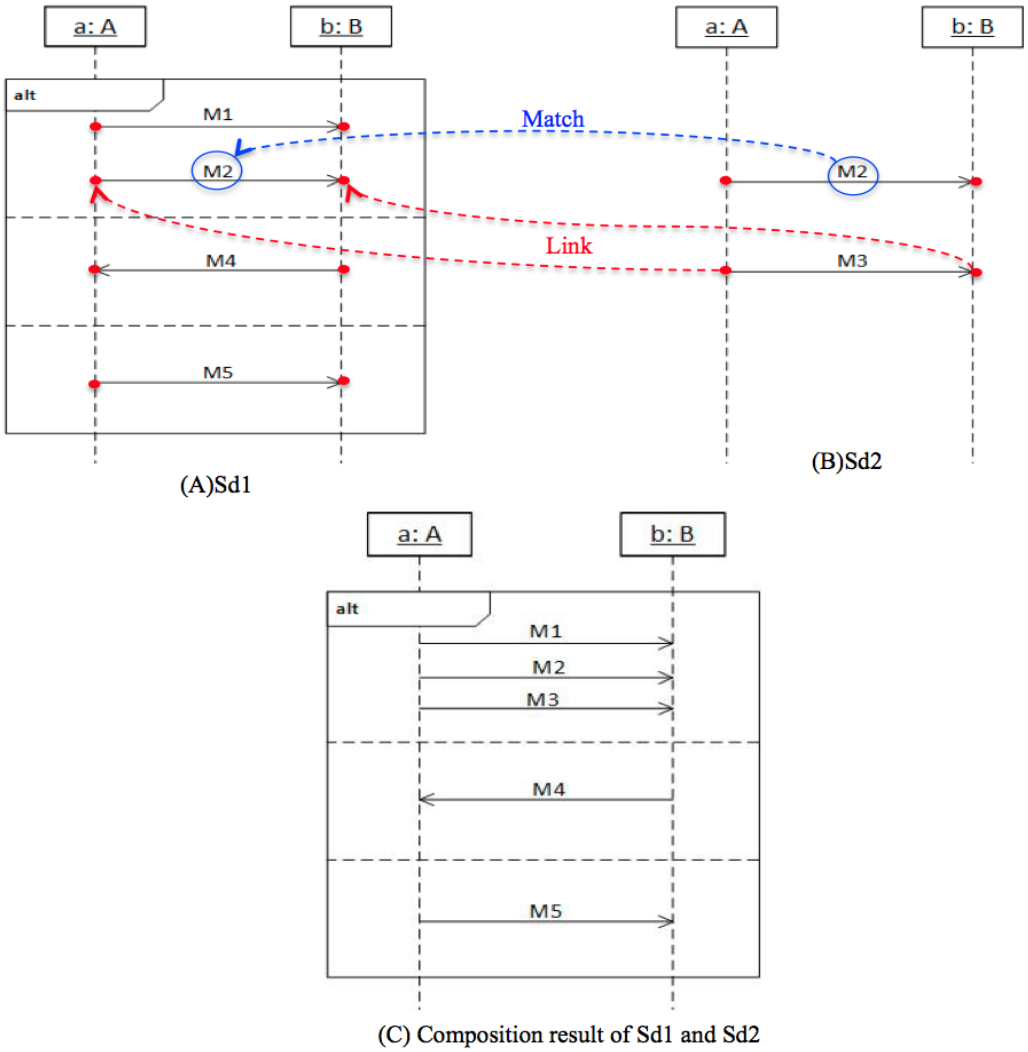


Figure 4.12: A composition example of sequence diagrams with CombinedFragments

The cases shown in Figure 4.12 consist of two sequence diagrams, each containing two lifelines and two messages. The sequence diagram, Sd1 as shown, contains an alternative CombinedFragment. Messages $M2$ in both diagrams are matched. However, $M2$ in Sd1 is allocated in the CombinedFragment (Figure 4.12). In order to compose this case, two steps are required. First, the messages and lifelines of the two diagrams need to be matched and composed, which can be done using the composition procedure explained in the previous sections. Secondly, if

the CombinedFragment is an 'alt' type, as is the case illustrated in the example, then the occurrence of the list of sd2 messages, subsequent to the removed message, i.e.$M3$, must be linked with the occurrence of the message $M2$ of sd1, which replaces $M2$ of sd2. Hence, the $M3$ message that now follows the $M2$ of sd1 will not occur unless the $M2$ message occurs. This process can be carried out by making the cardinality of the $M3$ send and receive messages equal to the cardinality of the $M2$ send and receive,which allocated in the '*alt*' CombinedFragment. This is illustrated in the fact below:

```
fact {# sd1_M2.send =# sd2_M3.send and # sd1_M2. receive =# sd2_M2. receive}
```

However, for the case of 'par', there is no occurrence link, as the events belonging to *par* CombinedFragment will occur in all solutions, but in a different order. Thus, only the messages and lifelines are composed if the type of CombinedFragment is *par* in Figure 4.12-A.

For example, consider the diagrams, sd1 and sd2 (see Figure 2.5); the messages $M1$ and $M2$ are matched in both diagrams. When these are composed, as shown in the 'Message composition' section, the occurrence of the messages follows $M2$ of Sd2, such that $M4$ and $M5$ must be linked with $M2$ as it is allocated in '*alt*' CombinedFragment. However, the messages follow $M1$ such as new will not change its occurrence as it is allocated in '*par*' CombinedFragment. This is illustrated in the fact below:

```
fact {
      #sd1_M2.send = #sd2_M4.send and #sd1_M2.receive = #sd2_M4.receive
      #sd1_M2.send = #sd2_M5.send and #sd1_M2.receive = #sd2_M5.receive}
```

There are other scenarios where both diagrams consist of two CombinedFragments. This case is shown in Figure 4.13, where two messages are matched from sd1, '$M1$' and '$M5$', and two with '$M1$' and '$M5$' of sd2. However, if the messages are composed, the solver will return *'unsat'*, due to the restriction of the metamodel, as shown in Figure 4.14.

The metamodel restriction will necessitate the composition of the InteractionOperands covering the message events and the CombinedFragment the InteractionOperand belongs to. This issue can be resolved by generating the fact which composes two CombinedFragments. The

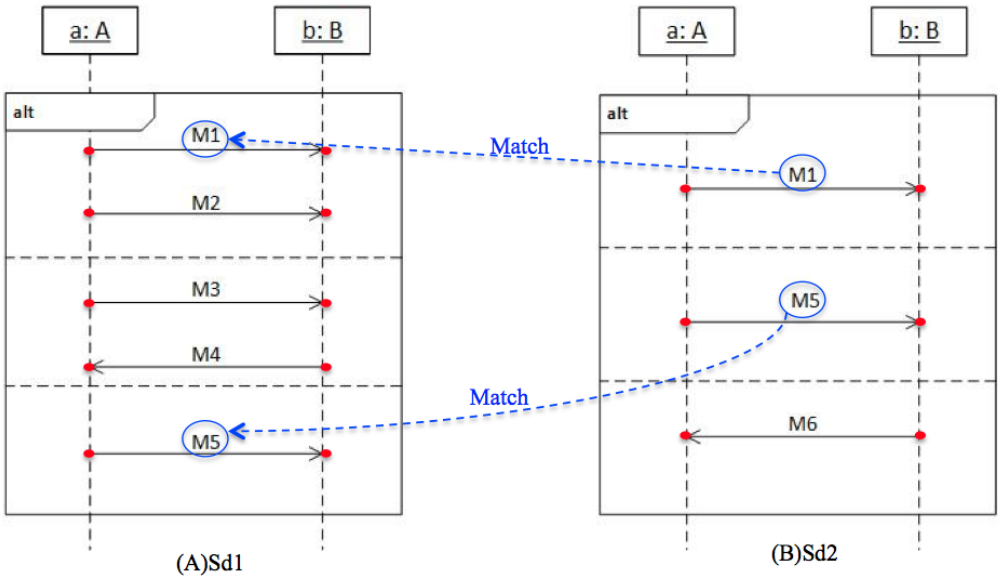


Figure 4.13: Sequence diagrams with matching CombinedFragments

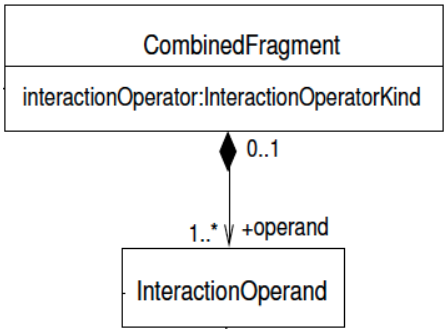


Figure 4.14: A CombinedFragment and InteractionOperand in the sequence diagram meta-model

matching condition in this case checks the operator types of CombinedFragments. If the operator types are the same (i.e. if both are alt), then the CombinedFragments will be able to compose, as the following code shows:

```
fact Combinedfragment { all CF1: sd1_Cf1 , CF1: sd2_Cf2 | (CF1.kind = CF2.kind ) =>#  CF2 =0
    and #sd2_CF2_Op1=0 and #sd2_CF2_Op2=0}
```

However, the limitation of the approach is revealed if the operators are different (i.e. if one is an '*alt*' and the other is a '*par*' type), Alloy Analyzer will return *'unsat'*. This is due to the inconsistent behaviour of the composed model (the operator types being different). Finally, the CombinedFragment of the sd2 consists of more than the composed InteractionOperands,

namely the InteractionOperand contains messages $M6$ with no match, as shown in Figure 4.13. Therefore, the InteractionOperand will be linked to the CombinedFragment of sd1.

4.3.1.4 Composition of the Running Example:

To evaluate this approach, the Alloy models were composed for sd1 and sd2 (see Figure 2.5). Alloy solutions for the composed model ($A3$), referred to as instances, were analysed. The solutions, shown in Figure 4.15, were compared with the LES model illustrated in Figure 2.9 and found to correspond to it. Figure 4.15, showing the Alloy instance consists of the messages and their events, where the messages are highlighted in red and their events are in black type. Moreover, the solution shows the order from top to bottom. Both instances demonstrate that '$m1$' occurs first and then 'i' is followed by 'j' or '$m2$' whereby the messages belong to the CombinedFragment, '*alt*'. A new message is shown in parallel with 'i' and 'j', but it always comes after '$m1$'. Finally, '$m3$' comes before '$m4$' in one instance and after '$m5$', in another, as Figure 4.15 shows. This is due to messages, '$m3$', '$m4$', and '$m5$' being parallel, as shown in Figure 2.9. However, note that whereas LES has a true concurrent semantics, traces in Alloy have an interleaving semantics, which means every instance shows a different order. The complete Alloy code for the running example is presented in Appendix B.

4.3.2 Behaviour Glue

This section describes the mechanism of a composition glue known as *behavioural glue*. The aim of this glue, as mentioned earlier, is to allow a designer to add further constraints to the composed model, in order to specify behaviour that should never occur (forbidden events), or sequences of events that must occur in a given order. This can be seen as a way of giving priority to certain specified interactions and eliminating some of the possible traces in the composed model. The behavioural glue described in Chapter 3, section 3.6.3 can be captured as *facts* in Alloy. All messages and their send and receive events belonging to a negative CombinedFragment are added to a fact called '*negativeTrace*'. In the body of this fact, the cardinality of the messages and their 'send' and 'receive' events are specified as '0'. Thus, the fact, *negativeTrace*

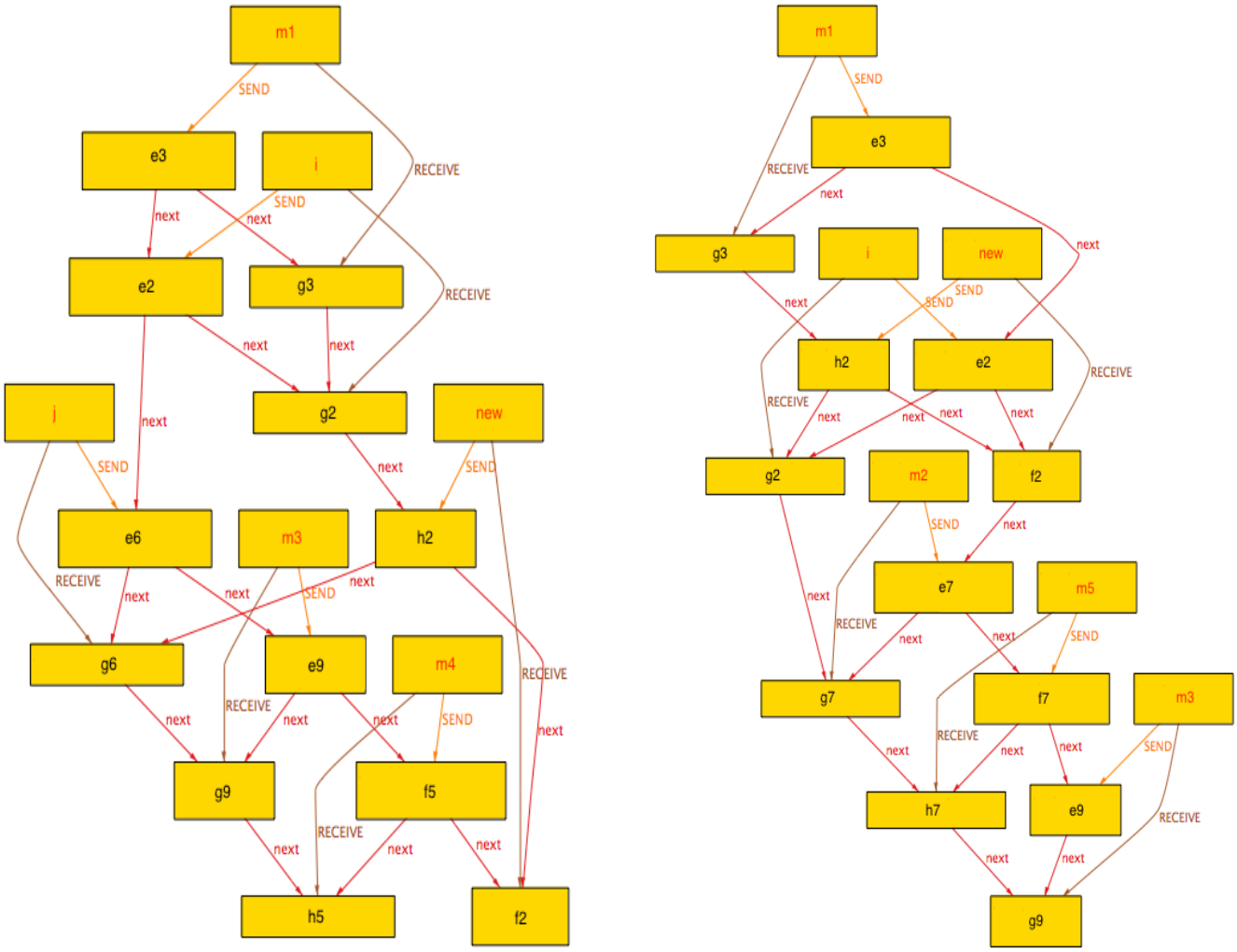


Figure 4.15: Examples of composition traces

will remove all messages and their events from the Alloy solution. The examples of behavioural glue introduced in Chapter 3, Figure 3.9 can be seen in the following facts:

```
fact negativeTrace {#sd1_j=0
all sd1_j_send:sd1_e6, sd1_j_receive:sd1_g6 |
#sd1_j_send=0 and #sd1_j_receive=0}
```

Sometimes, when forbidden messages are removed from a composed diagram, it becomes clear that some messages need to be reallocated, so that they occur in the optimal order. This means that some messages need to align their occurrences with the point following or preceding the occurrence of a specific message in the composed model. In consideration of the example in Figure 3.9, it is evident that the messages, $m2$ and j belong to an *alt* CombinedFragment and

the message, $m3$ will follow the CombinedFragment. As the example indicates j as a forbidden message, J will never occur in the Alloy instance after using the fact, *negativeTrace*, mentioned above (see Figure 4.16). However, as the message, $m2$ belongs to an alt CombinedFragment, it might not occur in all solutions, but $m3$ may occur in any solutions where the $m2$ message does not occur. This can be revealed as an incorrect result because the traces are affected by the forbidden events. Thus, after removing the forbidden message, it is essential to ensure that for $m3$ to occur, $m2$ must have previously occurred, in order to make sure that the whole solution presents a valid result. This can be encoded in a fact called an '*occurrence*'. The *occurrence* fact specifies that the cardinality of the message occurring first is equal to the cardinality of the message which subsequently occurs. Following this, it is specified that the order of the message and the underlying 'send' and 'receive' events that need to occur first, be followed by the 'send' and 'receive' events of the second message. The examples of behavioural glue introduced in Chapter 3, Figure 3.9 can be seen in the following facts:

```
fact negativeTrace {#sd1_j=0
all sd1_j_send:sd1_e6, sd1_j_receive:sd1_g6 |
#sd1_j_send=0 and #sd1_j_receive=0}

fact occurrence {
#sd1_m3.send =#sd1_m2.send and #sd1_m3.receive =# sd1_m2.receive
all sd1_m2_send:sd1_e7, sd1_m3_send:sd1_e9 |
sd1_m3_send in sd1_m2_send.^next
all sd1_m2_receive:sd1_g7, sd1_m3_receive:sd1_g9 |
sd1_m3_receive in sd1_m2_receive.^next}
```

The *negativeTrace* fact, above states that 'j' does not occur and moreover, neither do the associated events. The *occurrence* fact states that each time '$m3$' occurs, it must occur with '$m2$'. In other words, '$m2$' must occur first. Again, occurrence is controlled by the cardinality operator, $\#$. In addition, the behavioural glue for the 'occurrence' fact also defines the order of '$m3$' and the underlying 'send' and 'receive' events always come after the message, '$m2$'. The result of the above behavioural glue, above was checked with Alloy and the message 'j' does not occur in any solution obtained. Figure 4.16 shows two instances resulting from the

composition of the diagrams, sd1 and sd2, with respect to either glue. These instances represent the running of the traces in the semantic model, as shown in Chapter 3, Figure 3.8.

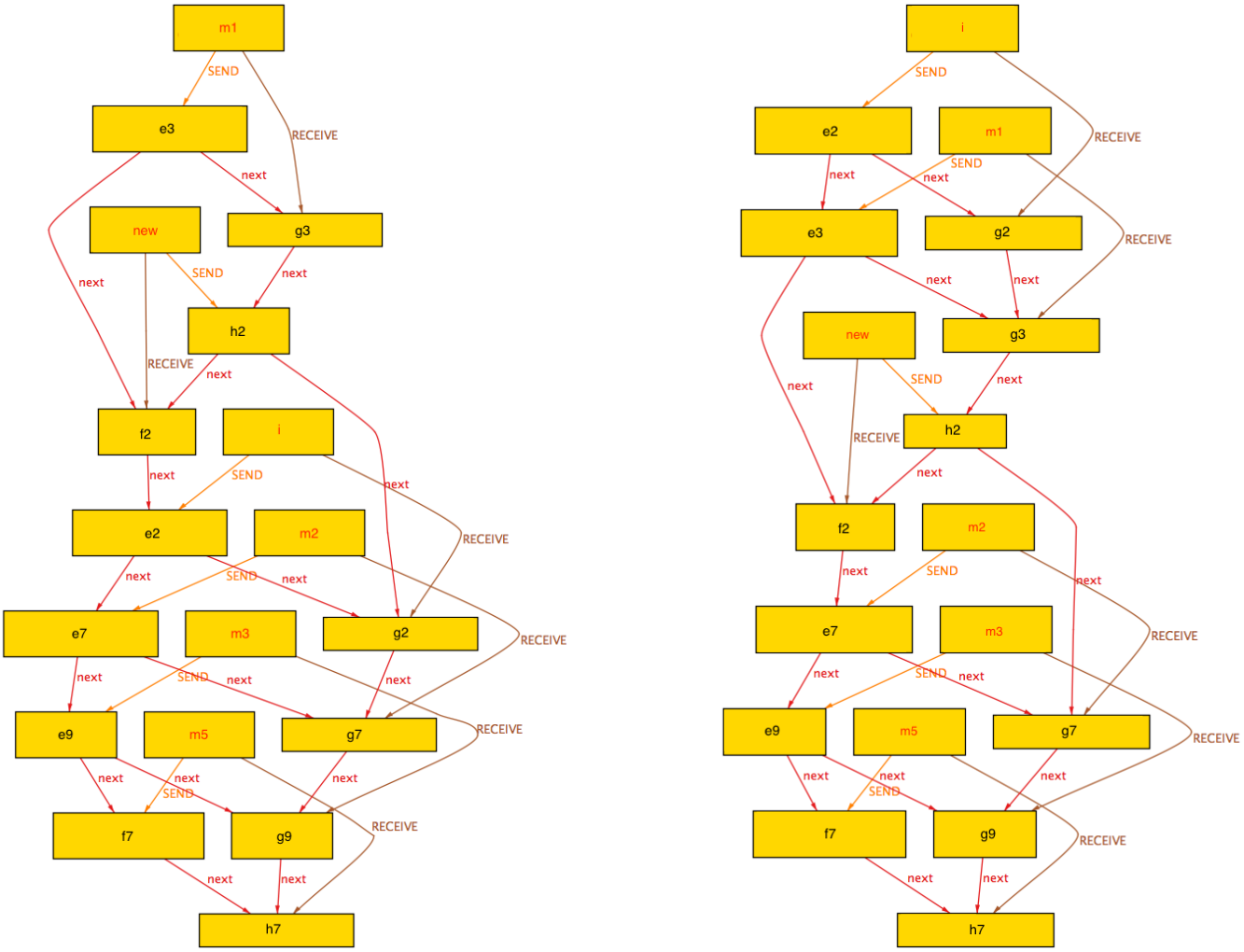


Figure 4.16: Examples of composition traces after removing message j

4.3.3 Limitations of the Approach:

In this chapter, an automated method of sequence diagram composition via Alloy has been presented. However, during the evaluation of this approach, Alloy revealed a performance issue when composing large sequence diagrams, whereby it can take hours to produce a solution and sometimes, the memory runs out. Consequently, only some cases of CombinedFragment composition have been considered in this chapter. Therefore, instead of considering more scenarios in Alloy, which will not guarantee that Alloy can handle them, it was decided to encode and

formalise the composition glue in Z3, as it is a state-of-the-art constraint solver. Hence, the Z3 SMT solver undertook the composition and the composition glue was formalised to be able to compose all possible scenarios, as shown in Chapter 5. In addition, Chapter 6 illustrates a comparison study, confirming the correction, whereby Z3 was chosen to compose complex sequence diagrams.

The transformation in this chapter illustrates seven transformation rules which transform part of the elements of a sequence diagram. However, some of the sequence diagram elements, such as an 'option' or 'loop' CombinedFragment are not covered in this approach. The option defined in [116] as a CombinedFragment represents a choice of behaviour, where either the InteractionOperand occurs, or else nothing occurs. The option is semantically equivalent to an alternative CombinedFragment, but contains only one InteractionOperand. The transformation of an option can be performed in the same way as the alternative CombinedFragment, but only one InteractionOperand will be generated for it instead of two, as stated in the definition. The occurrence option is known to be associated with a condition. This condition can be encoded as a fact, which will occur after the fact satisfies.

The loop operator specifies that all the messages forming part of its InteractionOperand are recurrent (looped) a specified number of times, based on the constraint attached to it; while still preserving the order between the messages. A loop CombinedFragment consists of only one InteractionOperand, but may contain other CombinedFragments. This CombinedFragment may then be transformed into a CombinedFragment signature with one InteractionOperand, similar to what is presented in the CombinedFragment rule. To model all possible iterations of the loop, the transformation needs to define the number of singleton signatures equal to the number of iterations, in order to represent the messages belonging to the loop. For example, if message '$m1$' belongs to a loop CombinedFragment that loops five times, then five message signatures are defined, namely '$m1\_1$', '$m1\_2$', etc. The first part of the name represents the name of the message ($m1$) and the second part represents the number of iterations, as Figure 4.17, below shows.

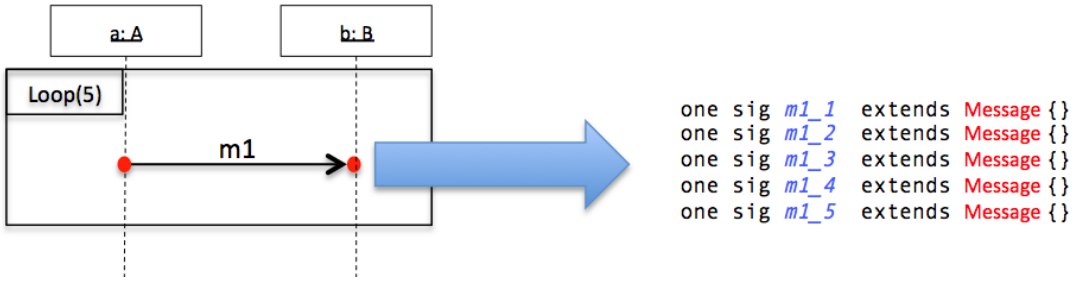


Figure 4.17: Finite loop in Alloy

4.4 Chapter Summary

Chapter 4 presents an automated method of sequence diagram composition via Alloy. The outline of the method involves the creation of logical constraints through EMR, which uniquely identify each component sequence diagram as an instance of the metamodel. To combine the models, logical constraints that compose the two models are produced. Some of these logical constraints declare matching elements and some enforce behaviour involved in the composition, such as specifying behaviour that should never occur, or sequences of events that must occur in a given order. This makes it possible for a designer to give priority to certain specified interactions, which are considered in the solution by eliminating unwanted traces from an initial matched model obtained. The result obtained automatically with Alloy preserves the formal interpretation of parallel composition with synchronisation glue. The model transformations presented in this chapter were implemented to automate the creation of logical constraints from sequence diagrams. Appendix A describes the implementation of this approach, which is developed as an eclipse plug-in called *SD2Alloy*.

CHAPTER 5

COMPOSITION OF SEQUENCE DIAGRAMS VIA Z3

5.1 Overview

In Chapter 4, a fully automated composition technique using Alloy was proposed. The approach does not, however, scale particularly well, especially when solving a complex model, which can take hours to produce a solution. To counteract this weakness, this chapter presents an alternative method of composition using the Z3-SMT solver. Z3 was selected as a target framework, due to it being a high-performance SMT solver, which can resolve the shortcomings of Alloy performance. In addition, the other advantage of using Z3 is that it is capable of displaying the overall model in a single solution, whereas Alloy produces as many solutions as there are possible traces in the model, with each solution representing a different trace.

In this chapter, there are two fundamental points that need to be considered when composing models: the mechanism must be well-defined and feasible for automation, and the composition algorithm must be efficient.

5.2 Sequence Diagrams in Z3

This section introduces the approach of transforming and composing sequence diagrams in Z3. As mentioned earlier, one of the aims of this approach is that mechanism must be well-

defined and feasible for automation. To address this point, the semantics of the models and their composition in the transformation is encoded to logical constraints, leaving the constraint solver to produce a solution for the composition, in accordance with these semantics. The second point, namely the efficiency of the mechanism, requires some further analysis by running a case study and various experiments, as well as making a comparison with suitable alternatives. Naturally, the problem arises when the models to be composed increase in size and complexity, but this is also influenced by how the transformation is implemented, the complexity of the composition algorithm, and the programming language used. However, in this chapter, the relevant approach is evaluated solely through a case study, i.e. a petrol station, whereas Chapter 7 presents a comparison study of the two approaches.

In Chapter 4, the approach adopted taken does not directly involve an algorithm to compose sequence diagrams, but rather uses Alloy to produce all possible solutions for the composition, where each solution is a possible trace of execution in the composed model. The composed model in Alloy satisfies the conjunction of all logical constraints underlying the models to be composed and additional matching constraints. The approach does not, however, explicitly incorporate the semantics of scenarios in the transformation itself. However, the approach in this chapter is more generic and covers a more complex form of composition.

As mentioned earlier in the overview, Alloy has shown some limitations in scalability. After closer inspection, the scalability problems would appear to be due to the fact that Alloy Analyzer, which underlies Alloy, is SAT-solver based and SAT-solving time may vary enormously, depending on factors such as the number of variables, the ordering of clauses and the average length of the clause [46]. Consequently, the time needed for analysis in Alloy will increase alongside its scope. Despite the fact this is more likely to be a problem inherent in Alloy and its implementation, another state-of-the-art solver will be explored in this chapter, namely Z3-SMT.

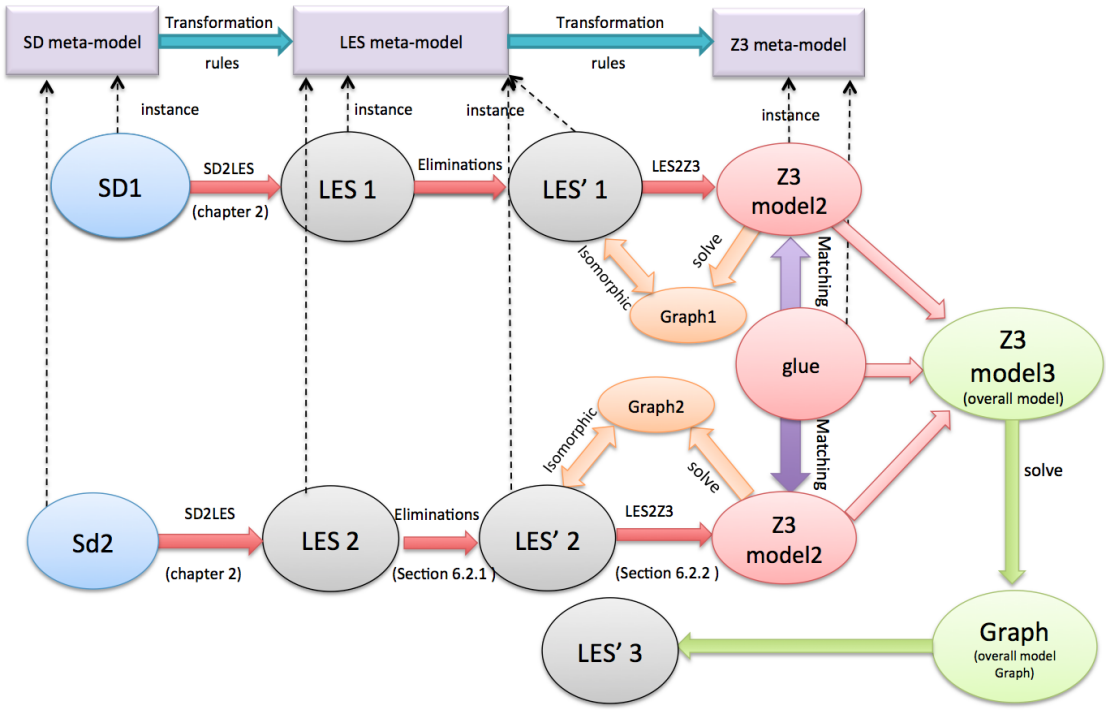


Figure 5.1: Composition approach in Z3

Figure 5.1 presents an overview of the composition approach in Z3. The approach to composing sequence diagrams in this instance can be described as follows: sequence diagram models are transformed into a textual representation of their underlying semantics in LES (see Chapter 2, section 2.2.2). Next, the LES models are reduced into LES' and transformed into equivalent Z3 models. The transformation is defined at metamodel level, basically obtaining a metamodel representation of sequence diagrams and LES, and translating elements of one metamodel into elements of the other. The transformation of LES' essentially consists of a set of events, relations and additional labels, which connect the LES' model and sequence diagram. All of these elements are defined in Z3. Thus, a unique Z3 model is produced for each LES' model, which, if solved, will produce a solution as a graph. This graph is isomorphic with the original LES' model, as illustrated in Figure 5.1 (see section 6.2.4).

After the above, a set of logical constraints representing the composition glue is produced. These constraints match the common elements of the input models. The constraint solver then solves these logical constraints and gives a solution corresponding to the augmented model, in accordance with the semantics of parallel composition. If the match cannot be made, Z3 will

return *'unsat'* (unsatisfiable), which means that no solution exists.

In the present approach, all models have been converted into Z3 specifications. This chapter focuses on the LES' to Z3 transformation step, whereas the transformation from sequence diagram to LES has been explained in Chapter 2, section 2.2.2. The mapping between a source (LES') and target (Z3) metamodel is defined by transformation rules, executed by a transformation engine for the source model (acting as input), in order to generate its equivalent target model (output).

5.2.1 Eliminating LES Model Events, except Message Send/Receive

In this approach, as mentioned in the previous section, LES model events are eliminated, except message send/receive events. Consequently, all the events that are not send or receive are removed, such as the beginning and end of an CombinedFragment, or the initial event of the lifeline. The aim of this is to reduce the size of the LES model and focus solely on the behaviour of model message events that represent the actual behaviour of the sequence diagram. This is for the sake of simplicity, when analysing transformation and composition models. For example, the CombinedFragments of the sequence diagram affect the message events that belong to it, regardless of the beginning and end events and the CombinedFragments. The main goal of the beginning and end events and the CombinedFragments is merely to define the location of the CombinedFragments in the LES model.

Figure 5.2 illustrates the reduction of the LES model for the sequence diagram, sd1 (Figure 3.7), which has been used in this thesis as a running example. As previously stated, the reduction process removes the events and relations belonging to it - shown in red - while the 'send and receive' events which remain are indicated in blue (see Figures 5.2-A, B). Moreover, it is important to mention that the events removed do not affect the behaviour or mapping of the concrete source sequence diagram. This means the dynamic interpretation of the sequence diagram is preserved.

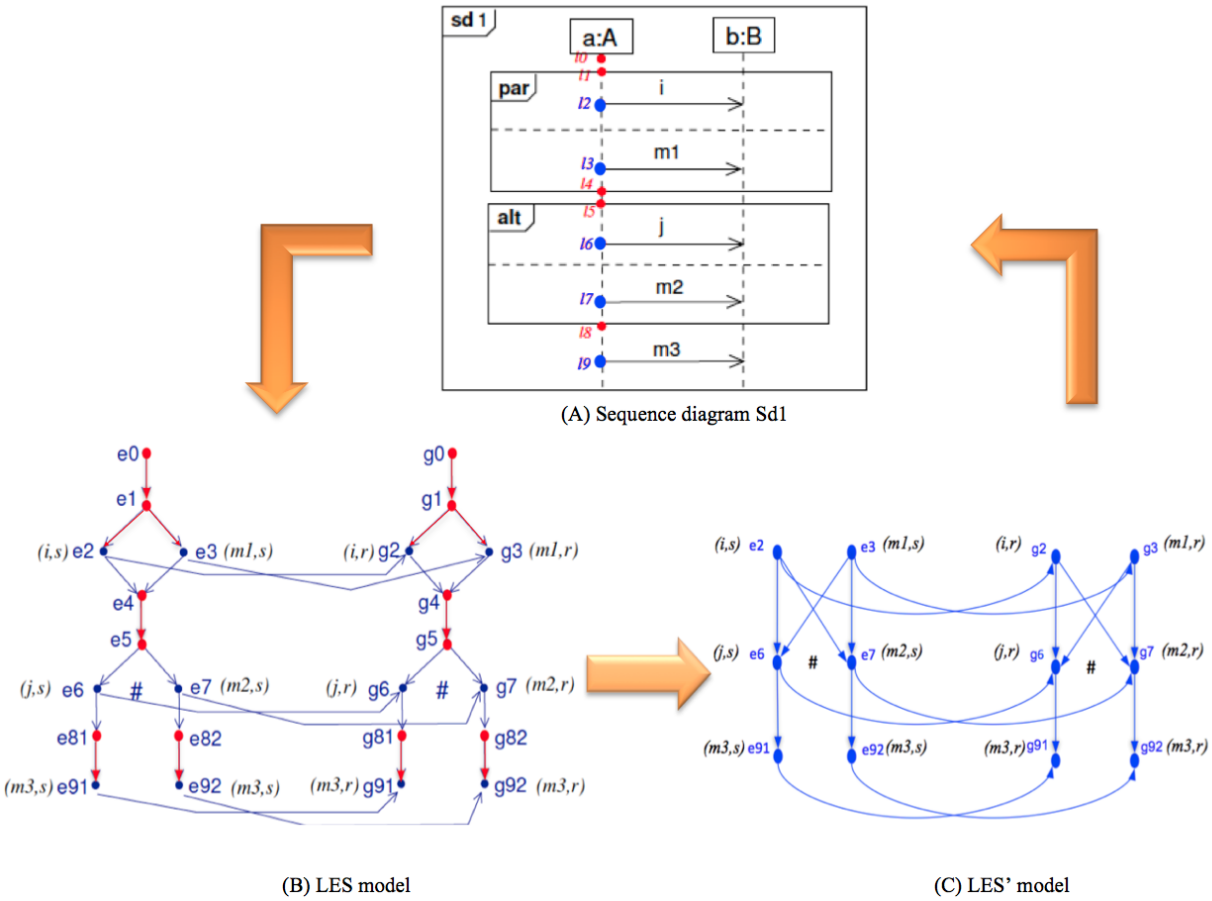


Figure 5.2: LES model and its LES'

5.2.2 LES2Z3: Model Transformation

As described in section 6.2, conducting a model transformation requires metamodels for the source and the target to be constructed to specify the elements of the source model that will be mapped to the elements of the target model. These are the source (LES) and the target (Z3-SMT) metamodels.

First, some of the main notations of Z3s syntax are recalled, which will be apparent in the following rules. Z3 supports many types of declaration, e.g. Integer, Real and Boolean, as well as allowing users to declare new sorts (types). The *DeclareSort* command is used to define the element domains, such as '*lifeline*', '*event*' and '*message*'. Furthermore, all the model elements are declared as a *'Constant'*, which is a function that does not accept any arguments. *Const(a,A)* is written to declare a constant, 'a' of type A.

Functions in Z3 are the basic building blocks of SMT formulas. They have no side effects and are total (i.e. they are defined for any element in the domain). Functions are used in this approach to define the relationship between elements, namely *causality, conflict*, etc. Finally, Z3 supports Boolean operators, such as *And, Or, Not, Implies* (logical implication), and equality == (bi-implication), among others. Universal (*ForAll*) quantifiers are also supported by Z3. In general, expressions in Z3 are built using set theoretical relational operators and constants.

Table 5.1 shows the mapping between the main concepts of LES (including labels) and Z3. In particular, LES is understood here as the semantic model for sequence diagrams, as discussed in Chapter 2, section 2.2.2. All main elements of the LES metamodel, such as events (Ev), lifelines (I) and messages (M) match a new type of element in Z3. This corresponds to creating new types called Ev, I and M using $DeclareSort$ (rules 1,3,6 in Table 5.1). Elements of these sets (as event, a message and a lifeline) are mapped onto constants of the corresponding sort (rules 2,4,7). The set of events in a LES used as a semantic model for sequence diagrams defines a partition determined by the set of instances I. This is dealt with in Z3 through a $cover$ function. In particular, if an event e belongs to an instance i_1 it cannot belong to a different instance i_2 (rules 5). A message is captured in an LES as a triple (e_1, m, e_2) such that $\mu(e_1) = (m, s)$ and $\mu(e_2) = (m, r)$ and is captured in Z3 as a function $isMsg$ that for a triple (e_1, m, e_2) determines whether it corresponds to a valid message tuple or not. A message always relates different events by causality (rule 8).

Furthermore, rules 9, 10 and 11 show how the binary relations between events in a LES are captured in Z3 and in accordance with the LES Definition 1. All relations are captured as functions in Z3 with additional constraints. The rules directly capture all the aspects of the formal definition given. For instance rule 9 shows how to define the partial order, that is, the relation is reflexive, antisymmetric and transitive. Rule 10 describes the conflict relation which is irreflexive, symmetric and propagates over causality. The concurrency relation in an LES (rule 11) represents an additional binary relationship between events. Rather than explicitly defining events in concurrency, any two events, which are not related by causality or conflict, are concurrent (more details on this will be given in the following rules).

Table 5.1: How LES for SDs are captured in Z3

	LES	**Z3**
1	Set of events Ev	Ev = **DeclareSort** ('Ev')
2	An event $e_1 \in Ev$	e1 = **Const** ('e1',Ev)
3	Set of instances or lifelines I	I = **DeclareSort** ('I')
4	An instance $i_1 \in I$	i1 = **Const** ('i1',I)
5	$Ev = \uplus_{i \in I} Ev_i$	cover = **Function** ('cover', Ev, I, BoolSort()) **ForAll**([e,i1,i2],**Implies**(**And**(cover(e,i1),(i1!=i2)), (**Not**(cover(e,i2)))))
6	Set of messages M	M = **DeclareSort** ('M')
7	A message $m \in M$	m = **Const** ('m',M)
8	For (e_1, m, e_2) $\mu(e_1) = (m, s)$ $\mu(e_2) = (m, r)$ and $e_1 \neq e_2$	isMsg = **Function** ('isMsg', Ev, M, Ev, BoolSort()) **ForAll**([e1,m,e2],**Implies**(isMsg(e1,m,e2),Next(e1,e2))) **ForAll**([e,m],(**Not**(isMsg(e,m,e))))
9	Causality $\rightarrow^* \subseteq Ev \times Ev$ is a p.o.: Reflexive Antisymmetric Transitive	Next=**Function**('Next',Ev,Ev,BoolSort()) **ForAll** ([e],(Next(e, e))) **ForAll**([e1,e2],**Implies**(**And**(Next(e1,e2),(e1!=e2)), **Not**(Next(e2,e1)))) **ForAll**([e1,e2,e3],**Implies**(**And**(**And**(Next(e1,e2), Next(e2,e3))),(Next(e1,e3))))
10	Conflict $\# \subseteq Ev \times Ev$ is irreflexive, symmetric, and propagates over $\rightarrow^*$	Conflict = **Function**('Conflict', Ev, Ev, BoolSort()) **ForAll**([e],(**Not**(Conflict(e,e)))) **ForAll**([e1,e2],**Implies**(**And**(Conflict(e1,e2),(e1!=e2)), Conflict(e2,e1))) **ForAll**([e1,e2,e3],**Implies**(**And**(**And**(Conflict(e1,e2), Next(e2,e3))),(Conflict (e1, e3))))
11	Concurrency e_1 *co* e_2 $\neg(e_1 \rightarrow^* e_2 \vee e_2 \rightarrow^* e_1 \vee e_1 \# e_2)$	Conc =**Function**('Conc',Ev,Ev,BoolSort()) **ForAll**([e1,e2],Conc(e1,e2)==**Not**(**Or**(Conflict(e1,e2), Next(e1,e2),Next(e2,e1))))

To keep it simple, only the transformation of the LES' for sd1 is shown in Figure 5.2.

5.2.3 Transforming Lifelines

As mentioned earlier, for each element in the metamodel, the transformation generates a *DeclareSor*. Thus, the transformation maps a lifeline element in the metamodel of the sequence diagram into *DeclareSor*, called l, as shown in Table 5.1, rule 3. Each concrete lifeline in the sequence diagram is mapped to a *Constant* (rule 4). Moreover, each lifeline object has a *name* and *class*, declared as a *Constant*. The link between the elements and their names and class can be specified using the functions referred to as *Lifeline_name* and *Lifeline_class*. For example, the lifelines in sd1 (Figure 5.3), as described in section 2.1, consist of two lifelines, to be transformed into the following Z3 code:

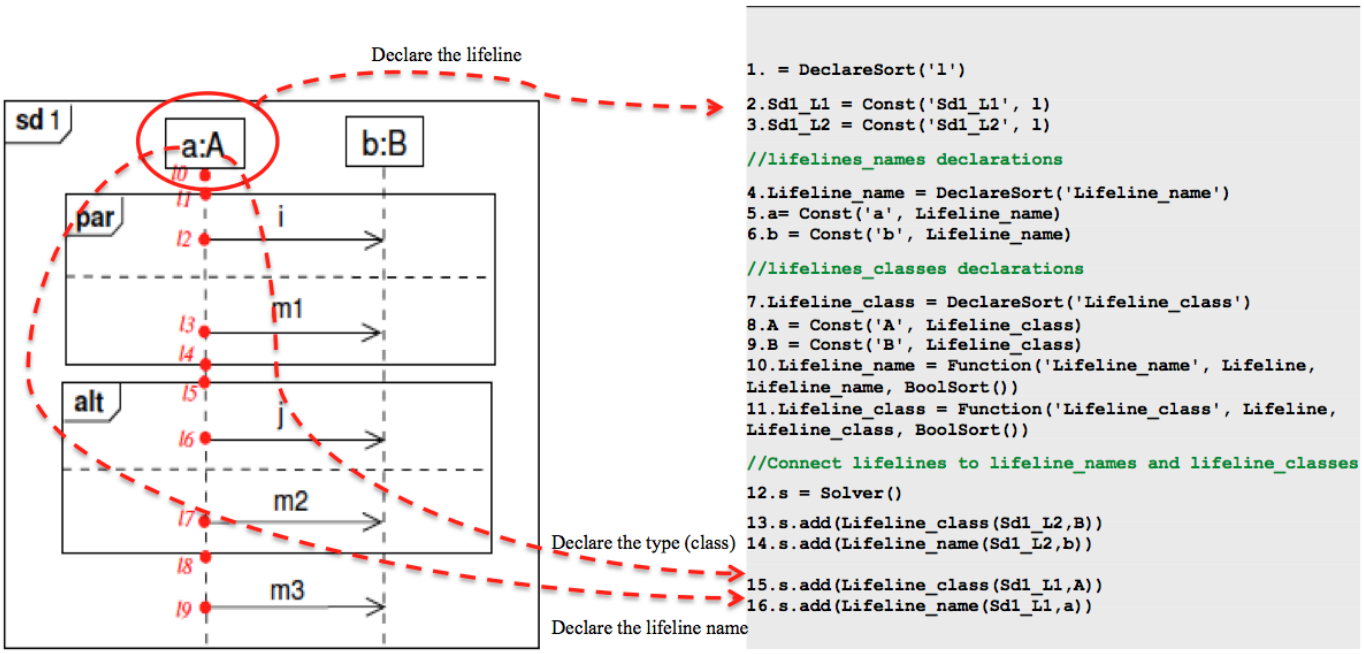


Figure 5.3: Lifeline declaration

Figure 5.3, above, shows the transformation of lifelines 'a' and 'b' in sequence diagram, sd1 into Z3 code. Line (1) shows the definition of the lifeline elements in the metamodel, which is called 'l'. Next, each concrete lifeline in the sequence diagram is declared as a *Constant* (lines 2-3). In addition, each lifeline object name and class are declared in lines 4-9, which are

a type of *Lifeline_name* and *Lifeline_class*. Lines 10-11 declare the functions, *Lifeline_name* and *Lifeline_class*. These functions are then used in lines 13-16 to assign the lifelines to their specific name and class.

Moreover, in Z3 it is possible to create a general purpose solver using *Solver()* , and to associate it to a particular variable by declaring *s=Solver()*. Constraints can be added using the method *add*.

5.2.4 Transforming Events

'Event' is used to represents the class, *OccurrenceSpecification* in the metamodel. Each event in the sequence diagram appears on precisely one lifeline, whereas a lifeline can have one or more events (as shown in the metamodel in Figure 2.4).

```
17. Ev = DeclareSort(Ev)
18. cover = Function('cover', Lifeline, Ev, BoolSort())
19. s.add(ForAll([L_i, e, L_j], Implies(And (cover(L_i, e) ,(L_i != L_j)), (Not(cover(L_j, e))
    ))))
```

The Z code, above, shows the declaration of the event element in the metamodel. Furthermore, the *cover* function in lines 17 defines the association between the lifeline and its events. The metamodel specifies that an *OccurrenceSpecification* (event) can appear on precisely one lifeline. This restriction has been defined in Z3 as an axiom (line 19). It shows that the lifeline can connect with many events, but each event is covered by at most one lifeline: formally, $Ev = \uplus_{i \in I} Ev_i$.

Similar to a lifeline, the transformation generates a *Constant* for each concrete event in the sequence diagram, as shown in Table 5.1, rule 2, whereas the function, *cover*is used to link the event to the lifeline that it belongs to. For example, the sd1 (Figure 5.2) consists of 12 events and is transformed into the following Z3 code:

```
20. Sd1_e2 = Const (Sd1_e2, Ev)
21. Sd1_e3 = Const (Sd1_e3, Ev)
22. Sd1_e6 = Const (Sd1_e6, Ev)
```

```
23. Sd1_e7 = Const (Sd1_e7, Ev)
24. Sd1_e91 = Const (Sd1_ e91, Ev)
25. Sd1_e92 = Const (Sd1_ e92, Ev)
26. Sd1_g2 = Const (Sd1_g2, Ev)
27. Sd1_g3 = Const (Sd1_g3, Ev)
28. Sd1_g6 = Const (Sd1_g6, Ev)
29. Sd1_g7 = Const (Sd1_g7, Ev)
30. Sd1_g91 = Const (Sd1_g91, Ev)
31. Sd1_g92 = Const (Sd1_g92, Ev)
```

The above code declares the sd1 events mapped from the LES' in Figure 5.2-C. The following function, *cover* connects the model events to the lifelines.

```
    //connect the events to lifeline a
32. s.add(cover(Sd1_a,Sd1_e2))
33. s.add(cover(Sd1_a,Sd1_e3))
34. s.add(cover(Sd1_a,Sd1_e6))
35. s.add(cover(Sd1_a,Sd1_e7))
36. s.add(cover(Sd1_a,Sd1_e91))
37. s.add(cover(Sd1_a,Sd1_e92))
    //connect the events to lifeline a
38. s.add(cover(Sd1_b,Sd1_g2))
39. s.add(cover(Sd1_b,Sd1_g2))
40. s.add(cover(Sd1_b,Sd1_g2))
41. s.add(cover(Sd1_b,Sd1_g2))
42. s.add(cover(Sd1_b,Sd1_g2))
43. s.add(cover(Sd1_b,Sd1_g92))
```

5.2.5 Transforming Messages

The transformation of messages creates *M*, a domain for messages, as shown in line 44 below. In the metamodel, a Message has two MessageEnds, namely a SendEvent and a ReceiveEvent. This is defined in a function $isMsg$ in line 45, as explained earlier. The constraint in line 46 determines that a single event cannot be a 'send' and 'receive' in the same message, which is formally defined in Table 5.1, rule 8. Moreover, in the metamodel of the sequence diagram, it is stated that a ReceiveEvent must always be preceded by a SendEvent. This rule, shown in line 47, which defines a constraint states that (e_i, m, e_j), $(e_i) = (m, s)$ and $(e_j) = (m, r)$, then

$e_i \rightarrow e_j$. Informally, message send events always occur before receive events. The constraint in line 47 shows the *iMNext* function. This represents the immediate causality relation. More information about this function is given later in the causality relation rule.

```
44. M = DeclareSort(M)
45. isMsg = Function ('isMsg', Ev, M, Ev, BoolSort())
46. s.add(ForAll([e_i,m],(Not(isMsg(e_i,m,e_i)))))
47. s.add(ForAll([e_i,m,e_j],Implies(isMsg(e_i,m,e_j),iMNext(e_i,ej))))
```

Similar to a lifeline, the transformation generates a *Constant* for each concrete message in the sequence diagram, as shown in Figure 5.4. Moreover, each message has $send/receive$ events, as illustrated in the following codes which define the messages of the example, sd1:

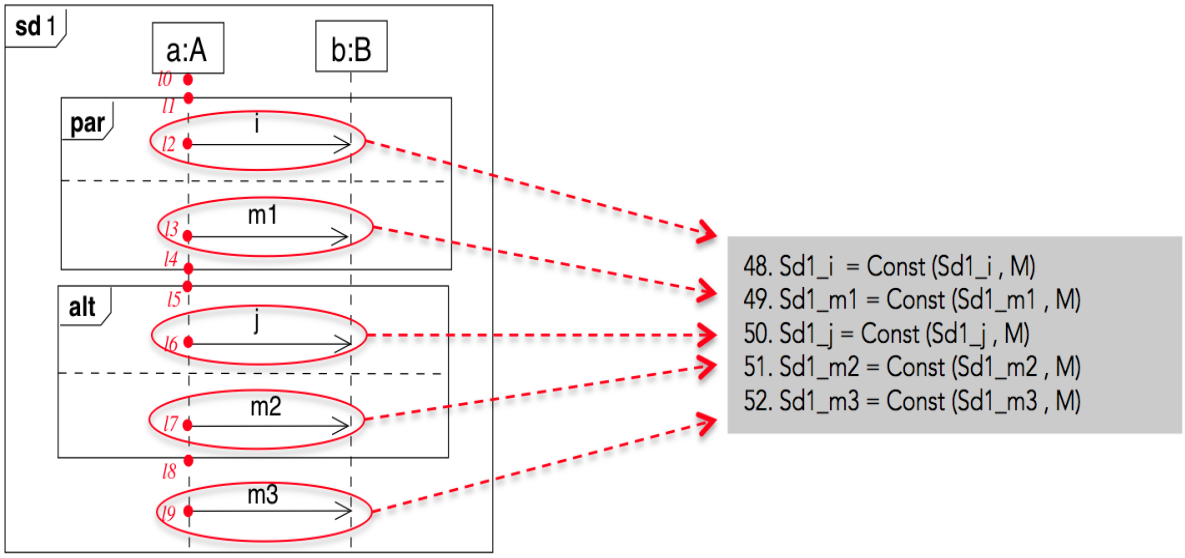


Figure 5.4: Message declarations

```
//connect messages and its events
53. s.add(isMsg1(Sd1_e2,Sd1_i,Sd1_g2))
54. s.add(isMsg1(Sd1_e3,Sd1_M1,Sd1_g3))
55. s.add(isMsg1(Sd1_e6,Sd1_j,Sd1_g6))
56. s.add(isMsg1(Sd1_e7,Sd1_M2,Sd1_g7))
57. s.add(isMsg1(Sd1_e91,Sd1_M31,Sd1_g91))
58. s.add(isMsg1(Sd1_e92,Sd1_M32,Sd1_g92))
```

The snippet of code above shows the assigning of send/receive events to their message, using the $isMsg$ function. Following the above labelling declarations, the next section will illustrate

the transformation rules for the LES relations.

5.2.6 Transforming the Causality Relation

A causality relation in LES represents a binary relationship between events. In general, it constitutes a partial order. Causality is specified in Z3 by introducing two Boolean functions, *Next* and *iMNext* (lines 59, 61). The function, *Next* represents actual causality ($\rightarrow^*$), whilst *iMNext* declares the immediate causality ($\rightarrow$) of all the sequence diagram events. Moreover, the constraints in lines 62-64 are aimed at obeying the metamodel restrictions of the LES. Thus, the causality is transitive, i.e. $e_i \rightarrow^* e_j$ and $e_j \rightarrow^* e_n$ then $e_i \rightarrow^* e_n$ for all $e_i, e_j, e_n \in Ev$, as specified in line 62. Moreover, the causality is antisymmetric, which means that for two events, $e_i \neq e_j$, such that $e_i \rightarrow^* e_j$ and then necessarily, $e_j \not\rightarrow^* e_i$. This is described in line 63, while line 64 shows that *Next* is reflexive.

Finally, the formula in line 65 states that all events are connected by immediate causality (*iMNext*) and actual causality (*Next*). The assertions in lines 66 and 67 show some sd1 events that are linked via the *iMNext* function, which are related through the immediate causality relation.

```
59. iMNext = Function('iMNext', Ev, Ev, BoolSort())
60. s.add(ForAll ([g_i],(Not(iMnext2(g_i, g_i)))))
// Actual causality
61. Next = Function('Next', Ev, Ev, BoolSort())
62. s.add(ForAll ([e_i,e_j,e_n], Implies(And(And(Next(e_i, e_j),Next(e_j, e_n))),(Next(e_i,
    e_n)))))
63. s.add(ForAll ([e_i,e_j], Implies(And(Next(e_i, e_j),(e_i != e_j)),Not(Next(e_j, e_i)))))
64. s.add(ForAll ([e_i],(Next(e_i, e_i))))
//All events connected by immediate Causality(iMNext) are connected by Causality relation (
    Next)
65. s.add(ForAll ([e_i,e_j], Implies (And(iMNext(e_i, e_j),(e_i != e_j)) ,Next(e_i, e_j))))
// adding immediate causality for the events
66. s.add(iMNext(Sd1_e0,Sd1_e1))
67. s.add(iMNext(Sd1_e1,Sd1_e2))
    ...
```

5.2.7 Transforming the Conflict Relation

A conflict relation in an LES represents a binary relationship between events. This relationship represents the behaviour of the alternative CombinedFragment, whereas each branch represents one interactionOperand of the CombinedFragment. In Z3, this is also specified by two new functions, *iConflict* and *Conflict* (lines 68, 69), which fulfill the same function as *Next* and *iMNext* in the causality relation. In addition to the direct conflict declared above, a constraint must also be included on the propagation of conflict over causality. This is formally defined in the LES, as follows: for events e_i, e_j, e_n if $e_i \# e_j$ and $e_j \rightarrow^* e_n$ then $e_i \# e_n$. Informally, it means that e_i is in conflict with e_j and e_n follows e_j then e_i has to be in conflict with e_n, which is specified in line 70, below. Line 71 states that the conflict function is symmetric, i.e. for two events $e_i \neq e_j$, such that $e_i \# e_j$ and then, necessarily, $e_j \# e_i$. Additionally, as specified in line 72, an event cannot be in conflict with itself (i.e. the relationship is irreflexive). The formula in line 73 states that all events connected by immediate conflict (*iConflict*) are also connected by conflict (Conflict). Finally, for events that are directly in conflict, constraints have to be imposed on the solver, as specified in lines 74 and 75.

```
68. iConflict=Function(iConflict,Ev,Ev, BoolSort())
69. Conflict=Function(Conflict,Ev,Ev, BoolSort())
70. s.add(ForAll ([e_i,e_j,e_n], Implies(And(And(Conflict(e_i, e_j),Next(e_j, e_n))),(Conflict
     (e_i, e_n)))))
71. s.add(ForAll ([e_i,e_j], Implies(And(Conflict(e_i, e_j),(e_i != e_j)),Conflict(e_j, e_i)))
    )
72. s.add(ForAll ([e_i],(Not(Conflict(e_i, e_i)))))
73. s.add(ForAll ([e_i,e_j], Implies (And(iConflict(e_i, e_j),(e_i != e_j)) ,Conflict(e_i, e_j
    ))))
// adding direct conflict
74. s.add(iConflict(Sd1_e6,Sd1_e7))
78. s.add(iConflict(Sd1_g6,Sd1_g7))
```

5.2.8 Transforming the Concurrent Relation

A concurrent relation in an LES represents a binary relation between events. This relation represents a parallel CombinedFragment. In Z3, this is specified as a new function called $Conc$.

```
Conc=Function(Conc,Ev,Ev, BoolSort())
```

The following constraint determines that rather than explicitly defining events in concurrency, any two events that are not related by causality or conflict, are concurrent. Therefore, there is no need to specify the events that are in a concurrent relation; the solver automatically generates this relation. The complete Z3 code for the running example is presented in Appendix C.

```
s.add(ForAll([e_i, e_j],Conc(e_i, e_j) == Not(Or(Conflict(e_i, e_j), Next(e_i, e_j),Next(e_j,
    e_i)))))
```

Z3 represents the solution as text, whereas the sequence diagram and LES model are visual. As a result, in order to check the validity of the solution, a parser has been implemented to map the Z3 solution to DOT language, which can then be executed using the *Graphviz* tool to produce the graph (Figure 5.5). Graphviz (Graph Visualization Software) is a package of open-source tools developed by AT&T Labs Research for drawing graphs specified in DOT language scripts [50].

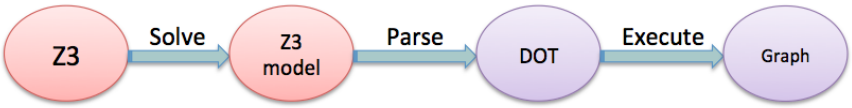


Figure 5.5: The parsing process

The snippet of code in Figure 5.6 illustrates a snapshot of the parser code. The parsing process can be briefly described in three steps. Firstly, the parser defines the name of the function, which needs to be parsed. Secondly, local variables contained in the function, are then replaced with actual names of the sequence diagram elements(Figure 5.7 -(B,C)). Thirdly,

```
def generate_events1(output):
    output = output[1:]
    output = output[:-1]
    result = ''

    import re
    pattern = "iMNext.*?]"
    next_section = re.findall(pattern, output, re.M|re.S)
    for x in next_section:
        if 'Event' in x:
            next_section = x
            break

    pattern = "\((Event.*?),.*?(Event.*?)\)"
    next_section_list = re.findall(pattern, next_section, re.M|re.S)


    output = re.sub('\[.*?\]', '___unused___', output, 0, re.M|re.S)


    next_section_list = str(next_section_list)

    pattern = '([a-zA-Z0-9_]+)\s.*?\s(.+),'
    for x in re.findall(pattern, output):
        next_section_list = next_section_list.replace("'"+x[1]+"'", "'"+x[0]+"'")

    next_section_list = eval(next_section_list)
    for x in next_section_list:
        result = result + "%s -> %s [label=%s]\n"%(x[0],x[1],'iMNext')
    return result
```

Figure 5.6: A snapshot of the parser code

the DOT code for the function is produced, which can be executed via the Graphviz tool to generate a graph, as Figure 5.7 -(D,E) shows.

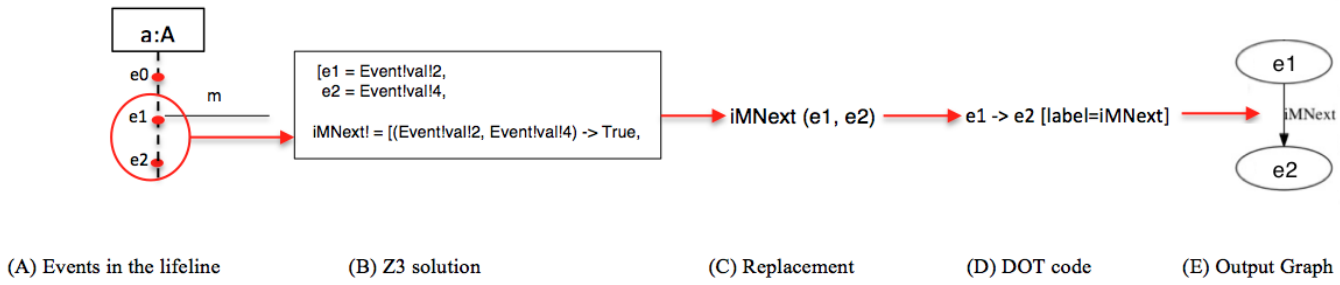


Figure 5.7: Example of a parsing mechanism

5.2.9 Isomorphism between a Z3 Graph and LES Model

In graph theory, the two graphs G and H graph are isomorphic if there is a *bijection* between the vertex sets of G and H:

$$f : V(G) \rightarrow V(H)$$

such that any two vertices u and v of G are adjacent in G if and only if $f(u)$ and $f(v)$ are

adjacent in H [37]. A *bijection* is a function mapping between the elements of two sets, where each element of one set is mapped with exactly one element of the other set, and each element of the other set is mapped with exactly one element of the first set.

There are multiple ways of proving a graph is isomorphic between the Z3 solution and LES' model to ensure the correctness of the transformation. The first of these involves mathematically proving the graph is isomorphic, whereas the second method involves using graph tools to automatically check the graph is isomorphic. As our approach mainly focuses on the practical side of model transformation, mathematical proof (the theoretical side) is outside the scope of this research. This is due to the fact that mathematical proof requires time, deep mathematical research and specific skills. Instead, the isomorphism between the graphs has been checked automatically in this instance, using the second method, namely a graph tool, such as R studio [124].

The R studio offers a package called $igraph$, which contains functions that map the two input graphs and determine whether they are isomorphic. The implementation of this package is based on the VF2 algorithm by Cordella et al.[37]. The procedure for checking isomorphism is as follows: firstly, the implementation in this instance automatically produced the solution graph in the GV extension. Secondly, both the Z3 graphs and the LES' model were converted as a *Graphml*. This is due to a tool, which accepts this *Graphml* extension as an input graph. The tool uploads these graphs to R and automatically compares them using the command, "graph.isomorphic.vf2" (graph1, graph2). The results consistently showed that the Z3 solution and LES' are isomorphic. Figure 5.8 presents LES model of sd1 (Figure 5.2) and its isomorphic graph produced by Z3.

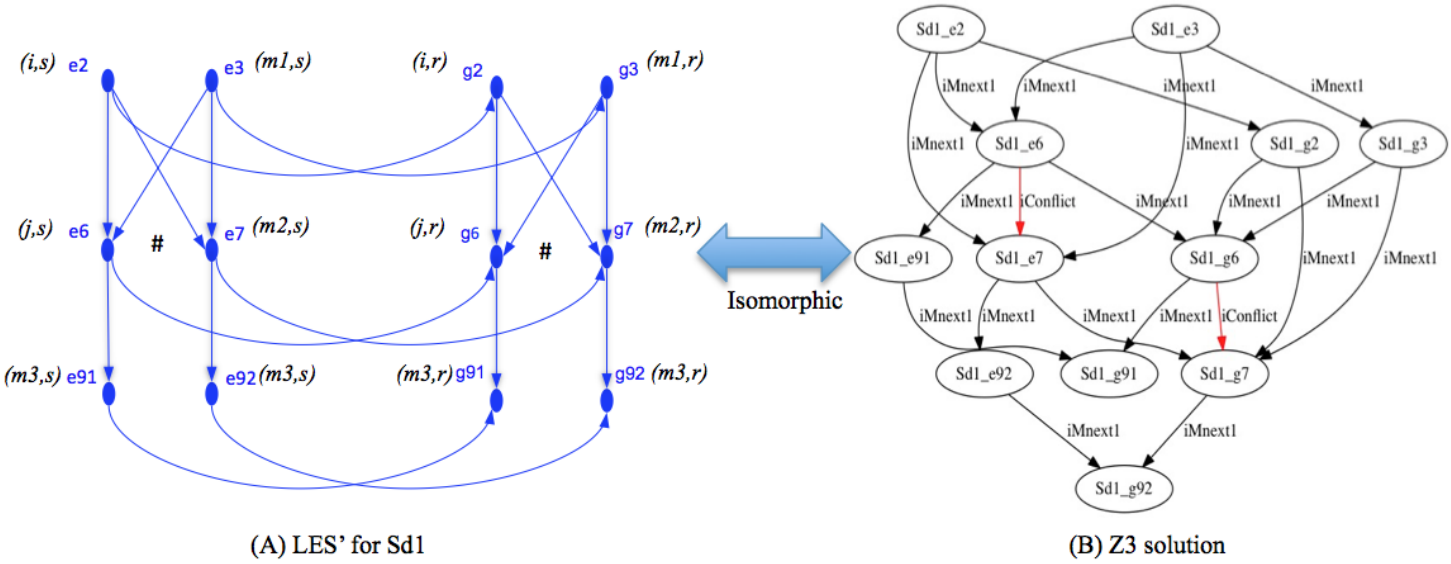


Figure 5.8: LES and Z3 solution

5.3 Composition of Sequence Diagrams in Z3

After the model transformation from LES' to Z3, the composition mechanism must be illustrated. Model3 (Z3 Model 3) in Figure 5.1 represents the composed model. This model consists of all the logical constraints of Z3-model 1, representing the LES' of sd1 and Z3-model2, representing the LES' of sd2 and the composition glue. The composition glue consists of three main functions that match the overlapping elements in the input, as shown below:

$$EventMatch(E_1, E_2) \rightarrow Bool$$

$$MessageMatch(M_1, M_2) \rightarrow Bool$$

$$LifelineMatch(L_1, L_2) \rightarrow Bool$$

The three lines above declare Boolean valued functions for the equality of model elements, i.e. messages, events and lifelines, respectively. The following sections explain the specification of the above functions in greater depth.

5.3.1 Specification of the Composition Glue

This section illustrates the specifications of the composition glue used to compose Z3 models. As mentioned earlier, the composition glue consists of three functions that have been designed to match the main elements of LES. Each of these functions is intended to match specific elements as the following subsections explain.

5.3.1.1 Event Match

The goal of the function, *EventMatch* is to match overlapping events in the input models. This function is a Boolean type function that matches two events from different models and returns '*true*' if the match satisfies the event matching axioms. Otherwise, it will return '*false*'. The function consists of a number of axioms, which specify how the models should be glued together to produce the intended composition. These axioms will be explained later in greater depth in section 6.3.2.

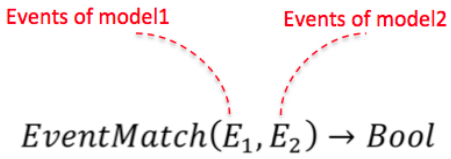


Figure 5.9: EventMatch function

The $EventMatch$ function is designed to only accept events as input as Figure 5.9 shows. This means it cannot match an event with another type of model element, such as a lifeline or message. For example, let us assume there are two diagrams and these consist of certain overlapping elements, as shown in Figure 5.10.

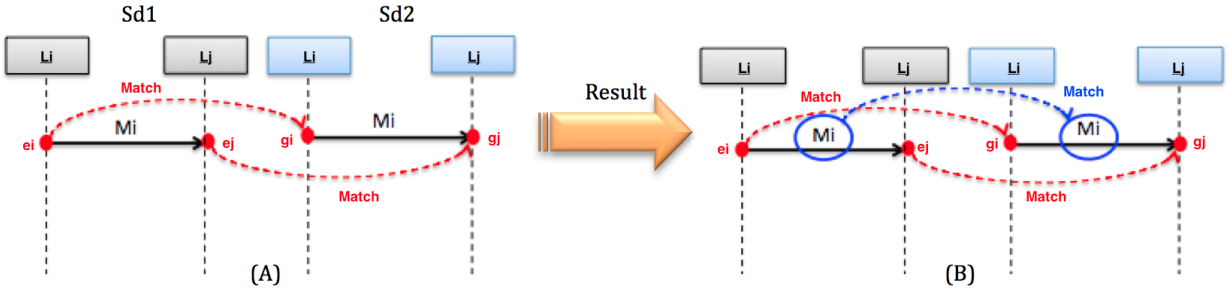


Figure 5.10: Simple diagrams with matched messages

In order to match the overlapping elements, i.e. messages 'Mi' in both diagrams, their events must first be matched, such as MessageSend and MessageReceive. This matching can only be conducted using the *EventMatch* function, thus $EventMatch(ei, gi)$ and $EventMatch(ej, gj)$.

5.3.1.2 Message Match

After the event match comes the *MessageMatch* function. This function is designed to match the overlapping messages from different diagrams. The form of the $MessageMatch$ is similar to the events function, which will only accept a message type of element as input. This function returns *'true*', if the message and its MessageSend and MessageReceive events match. As shown in the example in Figure 5.10, once the function $EventMatch$ return '*true*', the function $MessageMatch$ can be used to match the messages. In other words, the $MessageMatch$ function cannot be satisfied without matching the MessageSend and MessageReceive events of messages.

5.3.1.3 Lifeline Match

The purpose of the function, *LifelineMatch* is to match the lifelines in the models. Once the messages of the input models match, their lifelines will also match. This match is brought about by the $LifelineMatch$ function. Similar to the previous functions, the $LifelineMatch$ function is designed to exclusively match elements of the lifelines from the input models, as the Figure 5.11 shows.

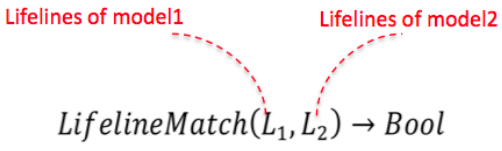


Figure 5.11: Lifeline Match function

As mentioned earlier, each of the above functions consists of a set of axioms that specify how the models should be glued together to produce the intended composition. The following section explains the axioms of each function in detail.

5.3.2 Composition Axioms and Cartesian Product Generation

The first step in the composition process is to pair the same type of input model elements, using a Cartesian product. Potentially, every element (of the same type) from Z3 model 1 can be paired with its corresponding type in Z3 model 2. For example, assuming (e1, e2) are events in sequence diagram 1, and (e3, e4) are events in sequence diagram 2, the Cartesian product of these events is as follows: {(e1, e3), (e1, e4), (e2, e3), (e2, e4)}. However, in this approach, the sole concern is to show the matching elements and the elements, which do not match. Therefore, the Cartesian product pairs are pruned. This means that only matching elements are displayed, whereas the elements, which do not match, are paired with a dash symbol (-). Thus, if it is assumed that if 'e1' matches 'e3', then neither 'e1' nor 'e3' are not permitted to pair with anything else. In this case, if the pair (e1, e3) exists as a matched pair, then the pairs (e1, e4) and (e2, e3) are removed. On the other hand, events that do not match such as e2 and e4 are paired as follows: (-, e4) and (e2, -), so that the pair (e2, e4) is also removed. Furthermore, as can be seen, the pairs (-, e4), (e2, -) contain a dash symbol (-). This symbol has been deployed to indicate that the element in a pair, which includes a dash, does not have any match. The result of pruning the above mentioned Cartesian product pairs is as follows: {(e1, e3), (-, e4), (e2, -)}. This illustrates that events (e1, e3) are matched, whereas the events (-, e4), (e2, -) do not have any matches in the other Z3 model. Finally, this result requires a function that can be used to display information about the elements of the composed model. Hence, we generate a function called *present*.

5.3.2.1 Present Function Technique

The goal of the *present* function is to display matching elements, as well as elements which are not matched in the composed model. The composed model referred to as Model 3 in Figure 5.1 consists of three kinds of present function, as follows:

$$EventPresent(E_1^1, E_2^1) \rightarrow Bool$$

$$MessagePresent(M_1^1, M_2^1) \rightarrow Bool$$

$$LifelinePresent(L_1^1, L_2^1) \rightarrow Bool$$

The first function, *EventPresent* is explicitly defined to present only the matched/unmatched events. For example, if the function *EventMatch* returns *'true'* for the pair of events mentioned earlier in this section, then *EventPresent* will display the events as a matched pair, i.e. *EventPresent* (e1, e3). Otherwise, *EventPresent* pairs the events that are not matched with a dash symbol (-), as follows:

$$EventPresent(e1, e3) \rightarrow true$$

$$EventPresent(-, e4) \rightarrow true$$

The same procedure applies to the lifelines (*LifelinePresent*) and messages (*MessagePresent*). To clarify the process of the '*present*' function, let us consider the following example:

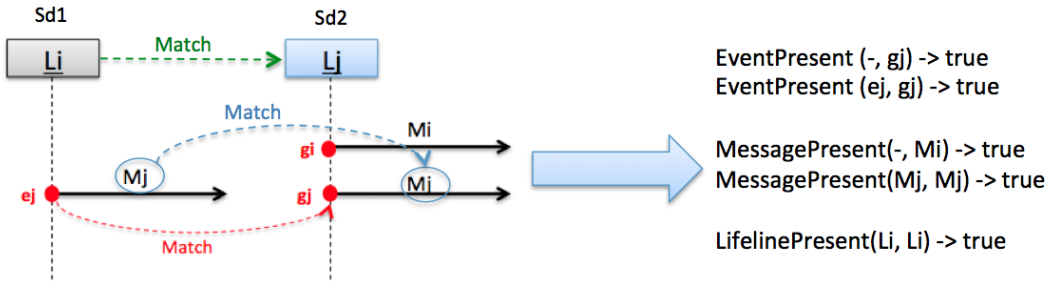


Figure 5.12: Representation of present function

The above Figure 5.12 shows two diagrams, sd1 and sd2. Sd2, consists of two messages, Mi and Mj. The message, Mj of sd2 matches the message Mj of sd1. Consequently, the send/receive events of both messages match and the lifelines these messages send and receive are also matched. Once we specify the matched events using the match functions illustrated earlier, the solver automatically produces present functions for the events, messages and lifeline, respectively. When the present function is generated, the information in the present function is used to display the information of the composed model elements, as the Figure 5.13 shows, below.

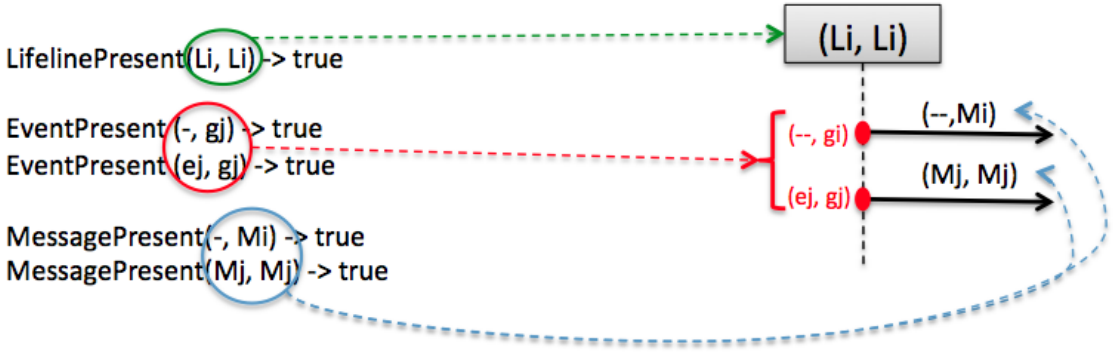


Figure 5.13: Information on present functions in the composed model

After displaying the result of the present functions, the following matching axioms are activated. In the following, there are 12 cases, which illustrate all possible matches between the events, messages and lifelines. Each case consists of a number of axioms with matching events, their messages and their lifelines, as the Table 5.2 shows.

Table 5.2: Composition cases

Cases		Descriptions
Events match in a causality relation	Case1	The axiom of Case 1 defines the matching of two events, connecting in a causality relation, with two other events, which are in a causality relation in a different model.
	Case2	The axiom of Case 2 defines the matching of one event in the first model, with an event in the second model that is followed by an event, which does not have any match.
	Case3	Case 3 defines the matching of one event in the first model with an event in the second model. However, the event in the first model is followed by an event, which does not have any match.
	Case4	This Case matches two events in different models, but the event in the first model is preceded by an event, which does not have any match.
	Case5	This Case is similar to Case 4, but the event in the second model is preceded by an event.
Events match in a conflict relation	Case6	The axiom of this Case defines the matching of two events in a conflict relation, with two others in a conflict relation and from a different model.
	Case7	The axiom of this Case defines the matching of one event in a conflict relation, from one model with another event in a different model.
	Case8	This Case is similar to Case 7, but it matches one event in a different branch of the conflict relation with an event in a different model.

	Case 9	This Case is similar to Case 7, but it matches one event in the second model, in a conflict relation with an event in the first model.
	Case 10	This is similar to Case 8, but the match is from the second model.
Parallel composition	Case 11 Case 12	These axioms define a parallel composition, if there are no matches between the events. Case 11 is designed for a parallel composition of events with their lifeline and Case 12, for events with their messages.

As Table 5.2 shows, the matched axioms are divided into three categories. The first category illustrates the cases relating to matches between events in a causality relation, whereas the second category shows cases relating to the matching of events in conflict relations. Finally, the third category demonstrates cases where there are no matches between events. Case 11 axioms specify the parallels composed between events and their lifelines, while Case 12 is written for parallel composition between events and messages. In the following, each case will be explained in greater depth.

- **Case 1:**

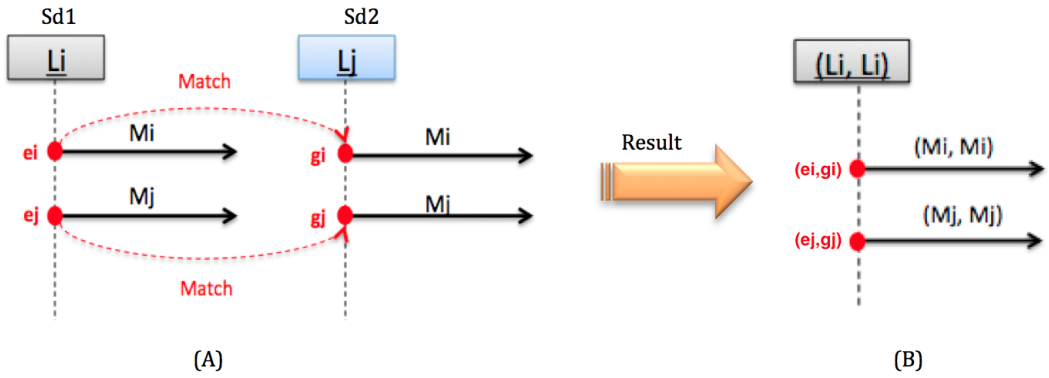


Figure 5.14: Case 1 scenarios

$$\forall e_i, e_j \in E_1, \forall g_i, g_j \in E_2|$$
$$iMNext_1(e_i, e_j)\&iMNext_2(g_i, g_j)\&EventMatch(e_i, g_i)\&EventMatch(e_j, g_j) \implies$$
$$iMNext_3((e_i, g_i), (e_j, g_j))$$

The above axiom aims to match the case where each diagram contains two pairs of events, one following the other: $iMNext_1(e_i, e_j)$, $iMNext_2(g_i, g_j)$, as shown in Figure 5.14-A. The functions, $iMNext_1$ and $iMNext_2$ refer to the causality relations of the sequence diagrams sd1 and sd2. These functions have been defined in section 6.2.4. The sd1 events match the sd2 events, i.e. $EventMatch(e_i, g_i)$ and $EventMatch(e_j, g_j)$. The composition then produces the function, $iMNext_3$, which represents the immediate causality relation in the composed model. The representation of the causality relation in the composed model is defined as follows:

$$iMNext_3((E_1^1, E_2^1), (E_1^1, E_2^1)) \rightarrow Bool$$

The above functions consist of two pairs of events next to each other. Each pair could be two matched events, i.e. (e_i, g_i), or unmatched events, i.e. $(e_i, -), (-, g_i)$. For example, Figure 5.15-B illustrates the result of this function, which contains the matched pairs $((e_i, g_i), (e_j, g_j))$. This indicates that the pair (e_i, g_i) comes before (e_j, g_j). Once the events are matched, their propagated matched axioms are automatically activated. Thus, their lifelines as well as their messages are matched, as the following axiom illustrates.

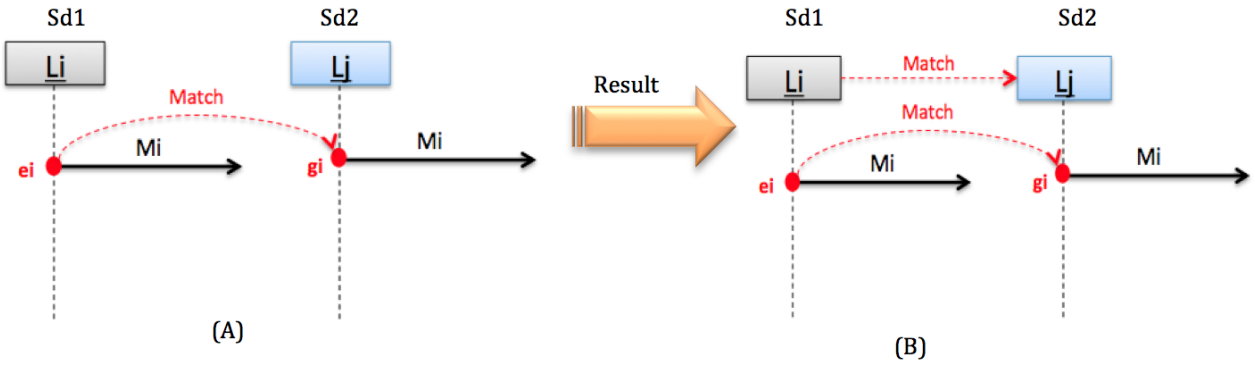


Figure 5.15: Matching lifelines

$$\forall e_i \in E_1, \forall g_i \in E_2, \forall l_i \in L_1, \forall l_j \in L_2|$$
$$EventMatch(e_i, g_i)\&cover_1(l_i, e_i)\&cover_2(l_j, g_i) \implies LifelineMatch(l_i, l_j)$$

The above axiom refers to the lifelines being matched. As can be seen, the axiom contains

the functions, $cover_1$ and $cover_2$ referring to the *cover* function relation of the models representing sd1 and sd2 (see Section 6.2.3). This axiom shows that if there is one event, $e_i \in E_1$ covered by lifeline l_i matching one event, $g_i \in E_2$ covered by lifeline l_j, then these lifelines are matched, as Figure 5.155-B illustrates. If the lifelines are declared as a matched, the following axiom creates the cover relation, $cover_3((l_i, l_j), (e_i, g_i))$.

$$\forall e_i \in E_1, \forall g_i \in E_2, \forall l_i \in L_1, \forall l_j \in L_2|$$
$$LifelineMatch(l_i, l_j) \& EventMatch(e_i, g_i) \& cover_1(l_i, e_i) \& cover_2(l_j, g_i) \implies$$
$$cover_3((l_i, l_j), (e_i, g_i))$$

$cover_3$ shows the association between the lifeline and the events in the composed model (model 3). This function is represented as follows:

$$cover_3((L_1^1, L_2^1), (E_1^1, E_2^1)) \rightarrow Bool$$

The above functions consist of two pairs $((L_1^1, L_2^1)$,$(E_1^1, E_2^1))$. The first pair (L_1^1, L_2^1) represents the matched/unmatched lifelines and the second pair represents the matched/unmatched events.

Moreover, in addition to the above axioms, syntactic matching is carried out. This match is performed using a function called *Lifeline Syntactic Matching*. The goal of this function is to compare the name and type (class) of lifeline that presented in section 6.2.2 and return it '$true$' if the lifelines match. Otherwise, the lifeline, which does not match, will be precisely specified.

In addition, the messages that these events belong to are also matched, as the following axiom shows:

$$\forall e_i, e_j \in E_1, \forall g_i, g_j \in E_2, \forall m_i \in M_1, \forall m_j \in M_2|$$
$$EventMatch(e_i, g_i) \& isMsg_1(e_i, m_i, e_j) \& isMsg_2(g_i, m_j, g_j) \implies$$
$$MessageMatch(m_i, m_j) \& EventMatch(e_j, g_j)$$

The above axiom explains the matching of messages. If the send events of messages, m_i, m_j are matched, then these messages are also matched, as well as the receive events. The same applies for receive events. Similar to the lifeline, the following axiom creates $isMsg3$ relation in the composed model, associated with the previous axiom. This function is represented as follows:

$$isMsg_3((E_1^1, E_2^1), (M_1^1, M_2^1), (E_1^1, E_2^1)) \rightarrow Bool$$

The above functions consist of three pairs $((E_1^1, E_2^1), (M_1^1, M_2^1), (E_1^1, E_2^1))$. The first pair represents the 'send' events, whereas the second pair represents the match/unmatched messages. Finally, the third pair represents the matched/unmatched 'receive' events, as the following axiom shows. The following axiom explains the case where events and messages are matched.

$$\forall e_i, e_j \in E_1, \forall g_i, g_j \in E_2, \forall m_i \in M_1, \forall m_j \in M_2|$$
$$MessageMatch(m_i, m_j) \& isMsg_1(e_i, m_i, e_j) \& isMsg_2(g_i, m_j, g_j) \implies$$
$$isMsg3((e_i, g_i), (m_i, m_j), (e_j, g_i))$$

The axiom shows that if two messages, m_i, m_j are matched, then they are composed and will produce, $isMsg3((e_i, g_i), (m_i, m_j), (e_j, g_i))$. The function, $isMsg3$ represents the association between the message and its events (send/receive) in the composed model. As can be seen, the axiom contains the function, $isMsg_1$ and $isMsg_2$ referring to the '$isMsg$ function' relation of the models representing sd1 and sd2 (see section 6.2.3). Similar to the lifeline, the messages are syntactically matched by comparing the names of the messages. This comparison is carried out using a function called *Messages Syntactic Matching*. This function compares the names of the messages and returns true if the messages match. The following snippet of code shows the above axioms written in Z3.

Z3 code for case 1:

```
//axiom for matching events.
ForAll ([e_i,g_i,e_j,g_j], Implies(And(And(iMNext1(e_i, e_j),iMNext2(g_i,g_j)),EventMatch(e_i,
     g_i), EventMatch(e_j, g_j)),iMNext3(e_i,g_i,e_j,g_j)))

//axiom for matching lifelines.
ForAll ([e_i,g_i,L_i,L_j], Imlies (And(EventMatch (e_i,g_i),cover1(L_i,e_i),cover2(L_j,g_i)),
    LifelineMatch (L_i,L_j)))

//axiom for generating cover3 that connect the match/unmatched events to match/unmatched
    lifeline in the composed model.
ForAll ([e_i,g_i,L_i,L_j], Implies (And(And(LifelineMatch(L_i, L_j),EventMatch(e_i,g_i)),
    cover1(L_i,e_i),cover2(L_j,g_i)),cover3(L_i, L_j,e_i,g_i)))
```

```
//axiom for matching messages.
ForAll ([e_i,e_j,g_i,g_j,M_i,M_j], Implies (And(EventMatch (e_i,g_i), isMsg1 (e_i,M_i,e_j),
    isMsg2(g_i,M_j,g_j)) ,And(MessageMatch (M_i, M_j), EventMatch (e_j, g_j)))))

//axiom for generating IsMsg3 that connect the match/unmatched events to match/unmatched
    message in the composed model.
ForAll ([e_i,g_i,e_j,g_j,M_i,M_j], Implies (And(And(MessageMatch (M_i, M_j)) ,isMsg1(e_i,M_i,
    e_j), isMsg2(g_i,M_j,g_j)),isMsg3(e_i,g_i,M_i,M_j,e_j,g_j)))
```

- **Case 2:**

The axiom of this case shows that the sequence diagram sd2 contains two events: $g_i, g_j \in E_2$, each following the other: $iMNext_2(g_i, g_j)$. Moreover, the first event, g_i matches the event, $e_i \in E_1^1$. In this case, the function, $iMNext_3$ consists of the matched pair (e_i, g_i) and the unmatched pair $(-, g_j)$. The function shows that the pair $(-, g_j)$ comes after the pair (e_i, g_i),which preserves the order of the original models, as illustrated in Figure 5.16-B.

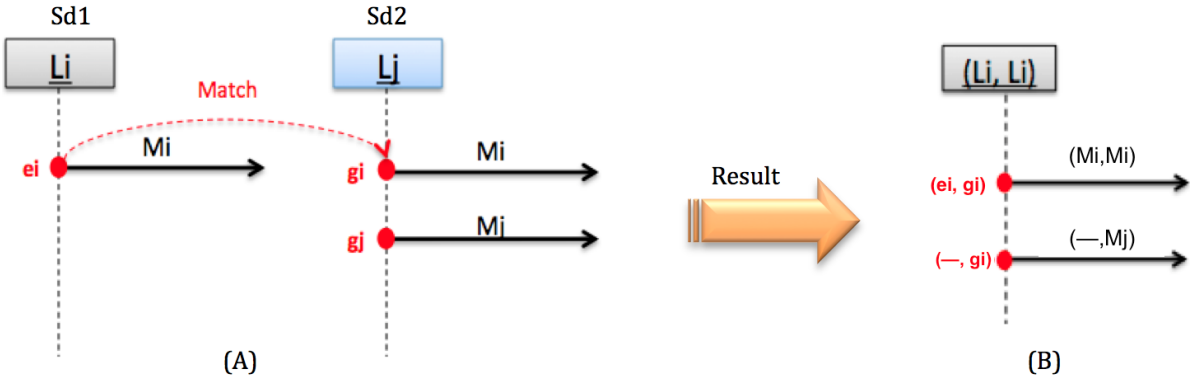


Figure 5.16: Case 2 scenarios

$$\forall e_i \in E_1^1, \forall g_i, g_j \in E_2 | EventMatch(e_i, g_i) \& iMNext_2(g_i, g_j) \& Notmatch_2(g_j) \implies iMNext_3((e_i, g_i), (-, g_j))$$

It must be noted that the above formula shows a function called $Notmatch2$. This function is a Boolean function representing events, which are unmatched. In other words, if the events are not assigned to the match function ($EventMatch$), the solver automatically assigns them to the $Notmatch$ function. In the composed model, there are two $Notmatch$ functions; one for the sd1 events, called $Notmatch1$ and one for the sd2 events, called $Notmatch2$.

The axioms propagated by the matched event are the same as those explained in case 1. However, for the unmatched event, the propagated axiom is as follows:

$$\forall g_i \in E_2^1, \forall l_i \in L_1, \forall l_j \in L_2 | LifelineMatch(l_i, l_j) \& Notmatch_2(g_i) \& cover_2(l_j, g_i) \implies cover_3((l_i, l_j), ((-, g_i))$$

The above axiom is an illustration of matched lifelines, which have some events that do not match any other events. In this case, the composed lifelines cover the events that do not match. The following snippet of code shows the axioms of Case 2 written in Z3 [1].

Z3 Code for Case 2:

```
//axiom for matching events.
ForAll ([e_i,g_i,g_j], Implies(And(And(EventMatch (e_i,g_i),iMNext2(g_i,g_j)),Notmatch2(g_j)),
     iMNext3 (e_i,g_i,empty1,g_j))))

//axiom for generating cover3 that connect the match/unmatched events to match/unmatched
    lifeline in the composed model.
ForAll ([g_j,L_i, L_j], Implies (And(LifelineMatch(L_i, L_j),cover2(L_j,g_j), Notmatch2(g_j)),
     cover3(L_i, L_j,empty1,g_j))).
```

[1] In Z3 code, the word $empty$ is used to represent the dash symbol (-)

- **Case 3:**

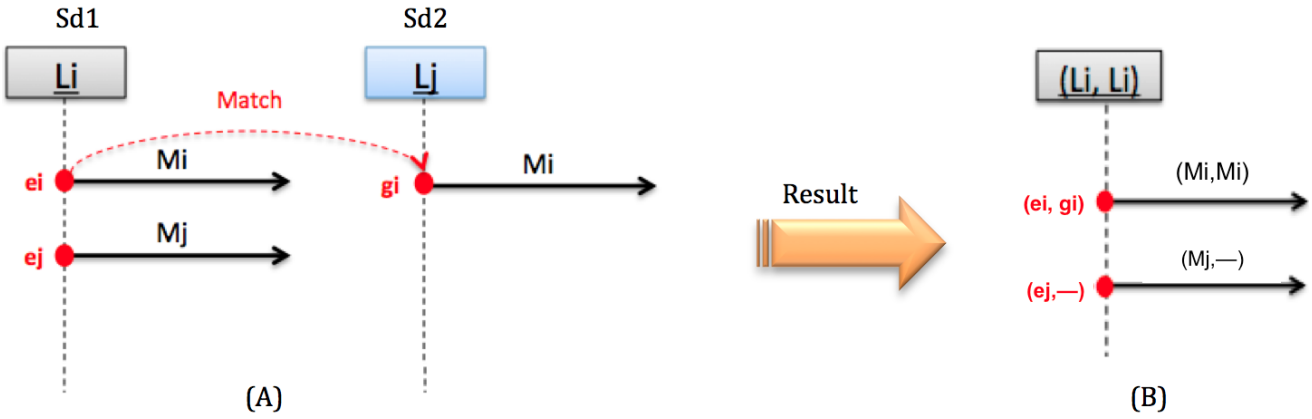


Figure 5.17: Case 3 scenarios

The axiom of Case 3 is similar to that of Case 2, but the first diagram event is followed by an event that does not have any match, as shown in Figure 5.17.

$$\forall g_i \in E_2^1, \forall e_i, e_j \in E_1 | EventMatch(e_i, g_i) \& iMNext_1(e_i, e_j) \& Notmatch_1(e_j) \implies iMNext_3((e_i, g_i), (e_j, -))$$

Z3 Code for Case 3:

```
ForAll ([e_i,g_i,e_j], Implies( And(And(EventMatch (e_i,g_i) ,iMNext1(e_i,e_j)) ,Notmatch1
    (e_j)),iMNext3 (e_i,g_i,e_j,empty2)))
```

- **Case 4:**

Case 4 explains the matching of two events (e_j, g_j) in diagrams sd1 and sd2. However, sd1 contains another event, e_i, which occurs before e_j. This event (e_i) does not match any event in sd2 (Figure 5.18-A). In this case, $iMNext_3$ consists of the two pairs, $iMNext_3((e_i, -), (e_j, g_j))$, which shows that $(e_i, -)$ comes before the pair (e_j, g_j), as shown in Figure 5.18-B. The propagated axioms in this Case are the same as those explained in Cases 1 and 2.

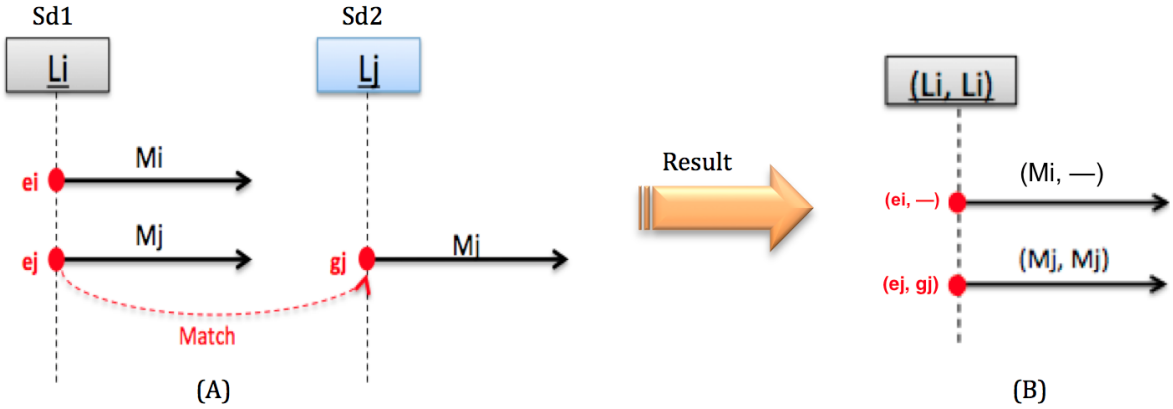


Figure 5.18: Case 4 scenarios

$$\forall g_j \in E_2^1, \forall e_i, e_j \in E_1 | EventMatch(e_j, g_j) \& iMNext_1(e_i, e_j) \& Notmatch_1(e_i) \implies iMNext_3((e_i, -), (e_j, g_j))$$

Z3 Code for Case 4:

```
//axiom for matching events.
ForAll ([e_i,g_j,e_j], Implies(And(And(EventMatch (e_j, g_j),iMNext1(e_i,e_j)),Notmatch1(
    e_i)),iMNext3 (e_i,empty2,e_j,g_j)))
```

- **Case 5:**

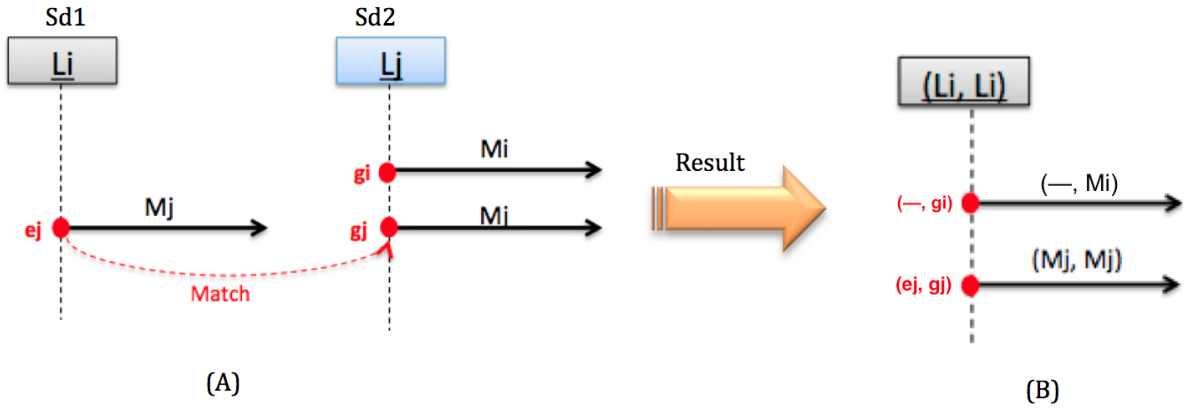


Figure 5.19: Case 5 scenarios

Finally, Case 5 is similar to Case 4, but here, $g_i \in E_2$ does not match any event in sd1.

$$\forall e_j \in E_1^1, \forall g_i, g_j \in E_2 | EventMatch(e_j, g_j) \& iMNext_2(g_i, g_j) \& Notmatch_2(g_i) \implies iMNext_3((-, g_i), (e_j, g_j))$$

Z3 Code for Case 5:

```
//axiom for matching events.
ForAll ([e_j,g_i,g_j], Implies(And(And(EventMatch (e_j, g_j),iMNext2(g_i,g_j)),Notmatch2(
    g_i)), iMNext3 (empty1,g_i,e_j,g_j)))
```

- **Case 6:**

This Case illustrates an instance of two sequence diagrams, each of which contains two events: $e_i, e_j \in E_1$ and $g_i, g_j \in E_2$ that are in conflict, as they belong to different interaction-Operands, $iConflict_1(e_i, e_j)$, $iConflict_2(g_i, g_j)$. The sd1 events match the sd2 events, namely $EventMatch(e_i, g_i)$ and $EventMatch(e_j, g_j)$. Thus, the composition produces a function; $iConflic_t3$ representing the conflict relation in the composed model. This function consist of two pairs, which shows that the matched pair (e_i, g_i) is in conflict with the matched pair (e_j, g_j),illustrated in Figure 5.20-C. Note that the composition of (e,g) was performed by the axioms in Case 1, as they are connected via a causality relation.

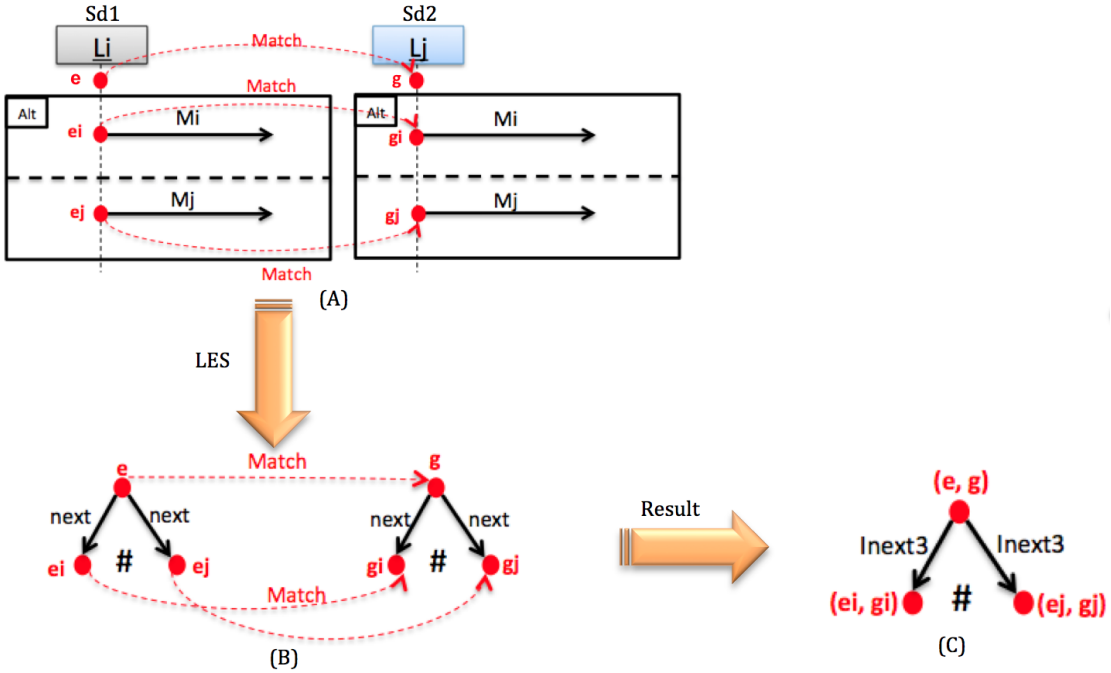


Figure 5.20: Case 6 scenarios

$$\forall e_i, e_j \in E_1, \forall g_i, g_j \in E_2|$$

$$iConflict_1(e_i, e_j) \& iConflict_2(g_i, g_j) \& EventMatch(e_i, g_i) \& EventMatch(e_j, g_j) \implies$$

$$iConflict_3((e_i, g_i), (e_j, g_j))$$

Z3 code for Case 6:

```
//axiom for matching events in conflict relation.
ForAll ([e_i,g_i,e_j,g_j], Implies(And(And(iConflict1(e_i, e_j),iConflict2(g_i,g_j)),
    EventMatch (e_i, g_i), EventMatch (e_j, g_j)),iConflict3(e_i,g_i,e_j,g_j)))
```

- **Case 7:**

This Case composes the scenario where two events, $g_i, g_j \in E_2$ contained in sd2 are in conflict, as shown in Figure 5.21-A, B. In this case, the first event (g_i) matches the event, (e_i) of sd1, whereas the event (g_j) does not have any matches. Thus, the function, $iConflict_3$ consists of two pairs $((e_i, g_i), (-, g_j))$, which shows the pair (e_i, g_i) are in conflict with the pair $(-, g_j)$, as indicated in Figure 5.21C.

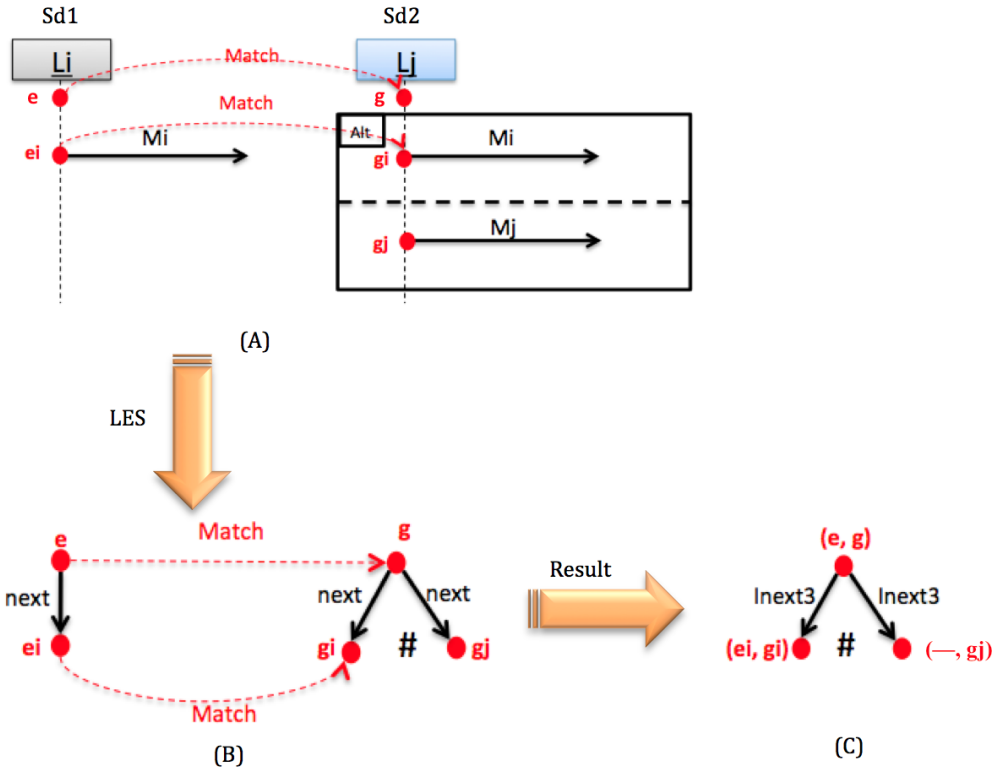


Figure 5.21: Case 7 scenarios

$$\forall e_i \in E_1^1, \forall g_i, g_j \in E_2 | EventMatch(e_i, g_i) \& iConflict_2(g_i, g_j) \& Notmatch_2(g_j) \implies$$
$$iConflict_3((e_i, g_i), (-, g_j))$$

Z3 Code for Case 7:

```
//axiom for matching events in conflict relation.

    ForAll ([e_i,g_i,g_j], Implies(And(And(EventMatch (e_i,g_i),iConflict2(g_i,g_j)),Notmatch2
        (g_j)), iConflict3 (e_i,g_i,empty1,g_j)))
```

- **Case 8:**

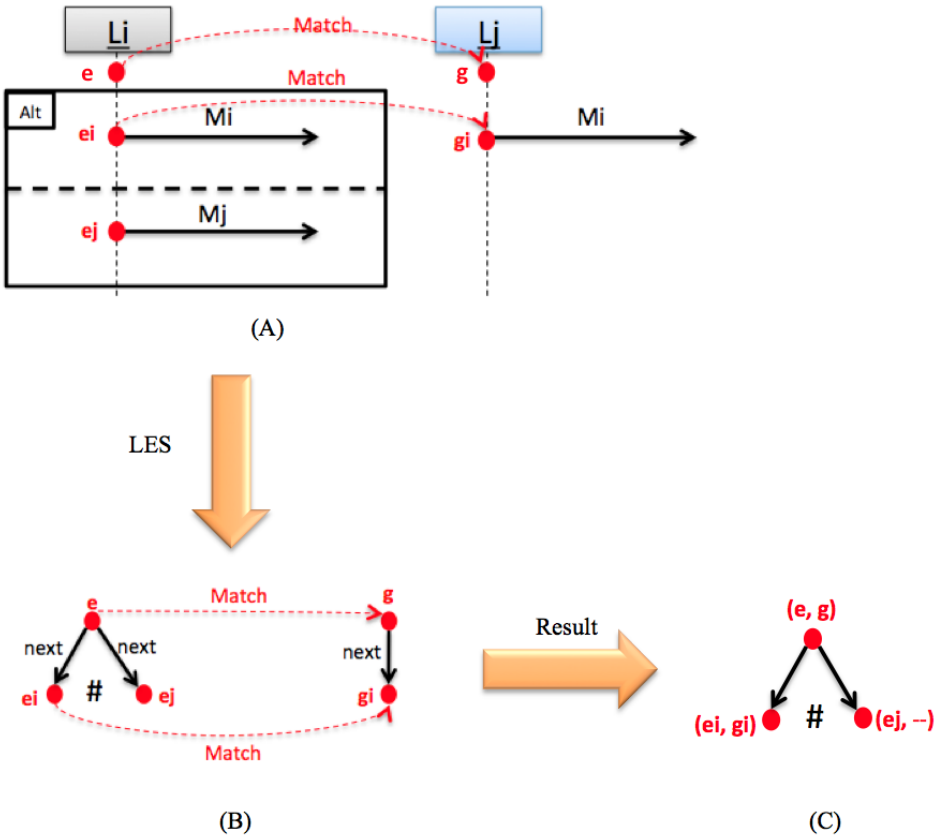


Figure 5.22: Case 8 scenarios

This Case is similar to Case 7, but here, the event (e_j), which in conflict with ei does not have any matches in the other diagram, as shown in Figure 5.22.

$$\forall g_i \in E_2^1, \forall e_i, e_j \in E_1 | EventMatch(e_i, g_i) \& iConflict_1(e_i, e_j) \& Notmatch_1(e_j) \implies$$
$$iConflict_3((e_i, g_i), (e_j, -))$$

Z3 Code for Case 8:

```
//axiom for matching events in conflict relation.
ForAll ([e_i,g_i,e_j], Implies(And(And(EventMatch (e_i,g_i),iConflict1(e_i,e_j)),Notmatch1
    (e_j)), iConflict3(e_i,g_i,e_j,empty2)))
```

- **Case 9:**

This Case considers the scenario where the second event (g_j) of sd2 matches the event (e_j) of sd1, whereas the event, g_i, which is in conflict with g_i does not have any matches. The resulting function, $iConflict_3$ shows that the matched pair (e_j, g_j) is in conflict with the unmatched pair $(-, g_i)$, as shown in Figure 5.23-C.

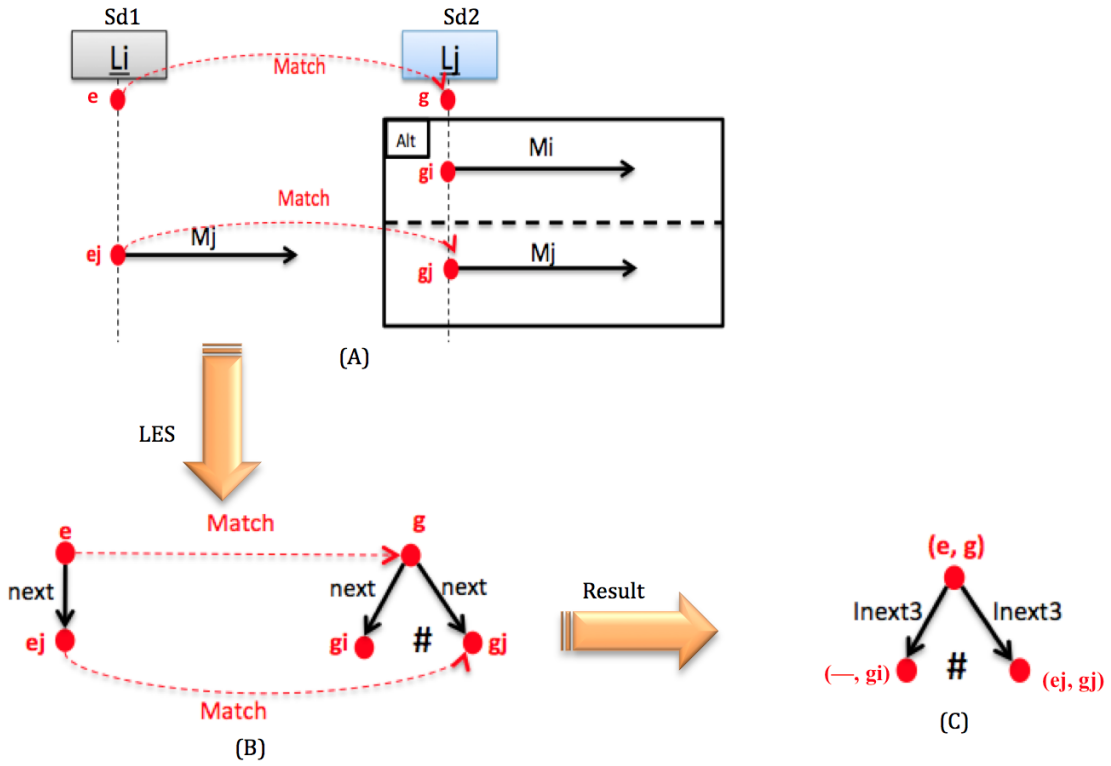


Figure 5.23: Case 9 scenarios

$$\forall e_j \in E_1^1, \forall g_i, g_j \in E_2 | EventMatch(e_j, g_j) \& iConflict_2(g_i, g_j) \& Notmatch_2(g_i) \implies iConflict_3((-, g_i), (e_j, g_j))$$

Z3 Code for Case 9:

```
//axiom for matching events in conflict relation.
ForAll ([e_j,g_i,g_j], Implies(And(And(EventMatch (e_j, g_j),iConflict2(g_i,g_j)),
    Notmatch2(g_i)), iConflict3(empty1,g_i,e_j,g_j)))
```

- **Case 10:**

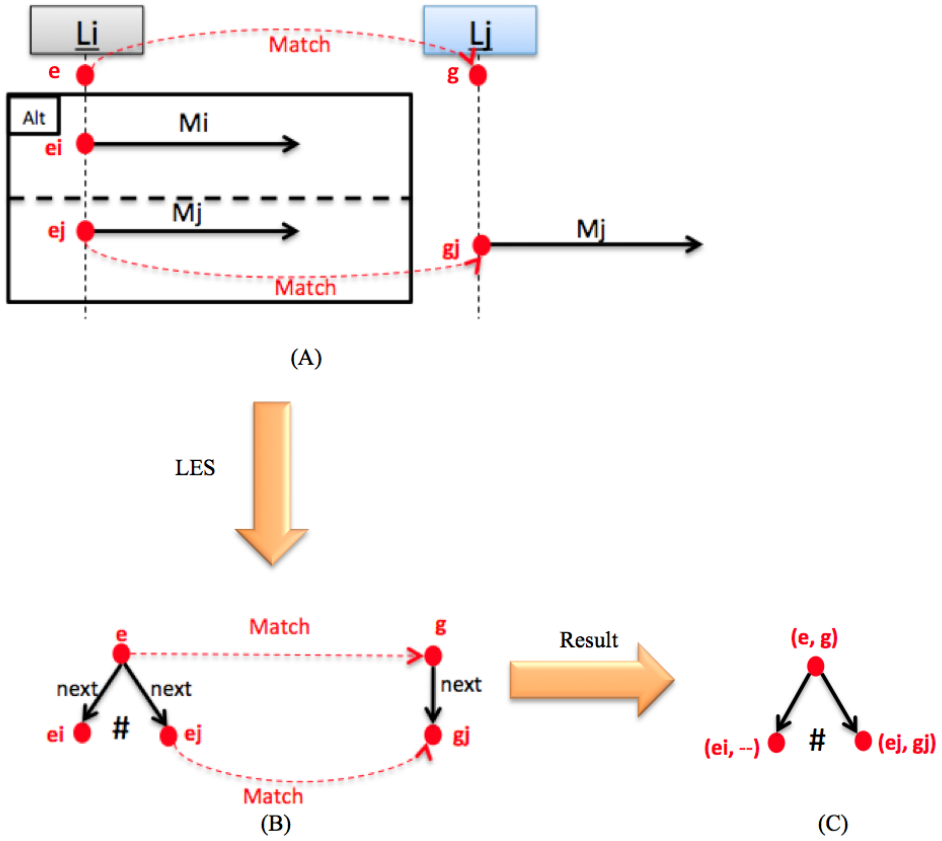


Figure 5.24: Case 10 scenarios

This Case is similar to Case 9, but here, the event (e_i), which is in conflict with ej does not have any matches, as shown in Figure 5.24.

$$\forall g_j \in E_2^1, \forall e_i, e_j \in E_1 | EventMatch(e_j, g_j) \& iConflict_1(e_i, e_j) \& Notmatch_1(e_i) \implies iConflict_3((e_i, -), (e_j, g_j))$$

Z3 Code for Case 10:

```
//axiom for matching events in conflict relation.

ForAll ([e_i,g_j,e_j], Implies(And(And(EventMatch (e_j, g_j),iConflict1(e_i,e_j)),
    Notmatch1(e_i)), iConflict3(e_i,empty2,e_j,g_j)))
```

- **Case 11:**

The following axiom produces a parallel composition. This axiom illustrates the case where the input models are not matched. Thus, the result shows two parallel lifelines and their events (Figure 5.25).

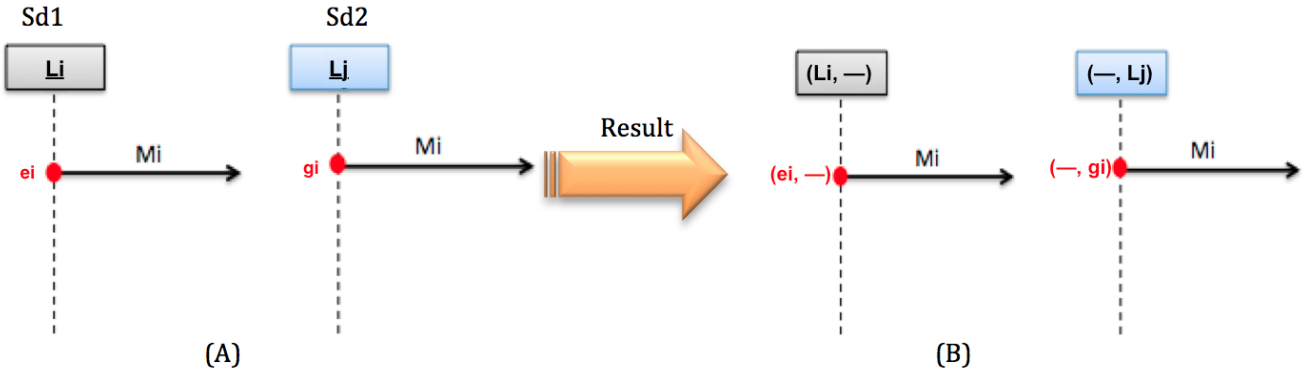


Figure 5.25: Case 11 scenarios

$$\forall e_i \in E_1^1, \forall g_i \in E_2^1, \forall l_i \in L_1^1, \forall l_j \in L_2^1|$$
$$cover_1(l_i, e_i)\&cover_2(l_j, g_i)\&Notmatch_1(e_i)\&Notmatch_2(g_i)\&LifelineNotMatch_1(l_i)$$
$$\&LifelineNotMatch_2(l_j) \implies cover_3((l_i, -), (e_i, -))\&cover_3((-, l_j),$$
$$(-, g_i))$$

Note that the above formula shows a function called $LifelineNotMatch$ in addition to the function $Notmatch$ explained earlier. This function is a Boolean function representing lifelines, which are unmatched. In other words, if the lifelines are not assigned to the match function ($lifelineMatch$), the solver automatically assigns them to the $LifelineNotMatch$ function. In the composed model, there are two $LifelineNotMatch$ functions; one for the sd1 events, called $LifelineNotMatch1$ and one of the sd2 events, called $LifelineNotMatch2$, as

the above formula shows.

Z3 Code for Case 11:

```
//axiom for parallel composition of the events and lifelines.
ForAll ([e_i,g_i,L_i, L_j], Implies (And(And(And(cover1(L_i,e_i),cover2(L_j,g_i)), Notmatch1(
    e_i), Notmatch2(g_i)),LifelineNotmatch2(L_j),LifelineNotmatch1(L_i)) ,And(cover3(L_i,
    empty4,e_i,empty2),cover3(empty3, L_j,empty1,g_i)))).
```

- **Case 12:**

The axiom in this Case produces a parallel composition for the messages and its send/receive events, which illustrates that the input models do not have any matches between the messages or their events. This axiom is associated with the axiom in Case 11.

$$\forall e_i, e_j \in E_1^1, \forall g_i, g_j \in E_2^1, \forall m_i \in M_1^1, \forall m_j \in M_2^1|$$

$$MessageNotMatch_1(m_i)\&MessageNotMatch_2(m_j)\&isMsg_1(e_i, m_i, e_j)\&isMsg_2(g_i, m_j, g_j)$$

$$\implies isMsg_3((e_i, -), (m_i, -), (e_j, -))\&isMsg_3((-, g_i),$$

$$(-, m_j), (-, g_i))$$

As can be seen, the formula contains a function called *MessageNotMatch*. This function plays the same role as the functions *Notmatch* and *LifelineNotMatch* explained in Case 11.

Z3 Code for Case 12:

```
//axiom for parallel composition for the messages.

ForAll([e_i,g_i,e_j,g_j,M_i,M_j],Implies(And(And(MessageNotmatch1(M_i),MessageNotmatch2(
    M_j)), isMsg1(e_i,M_i,e_j),isMsg2(g_i,M_j,g_j),And(isMsg3(e_i,empty2,M_i,empty6,e_j,
    empty2),isMsg3(empty1,g_i,empty5,M_j,empty1,g_j))))
```

To evaluate the glue, the above axioms have been applied in the running example shown in Figure 2.5. This example shows that the messages, $m1$ and $m2$ are the same in both diagrams, sd1 and sd2. Thus, the function, *MessageMatch* can be used to match the messages, such that: *MessageMatch* (sd1.m1, sd2.m1) = true, and *MessageMatch* (sd1.m2, sd2.m2) = true. Once the function is assigned, the propagated axioms for matching send and receive events and

their lifelines are automatically activated, in order to compose the overlapping elements. The complete Z3 code for the composition is presented in Appendix C.

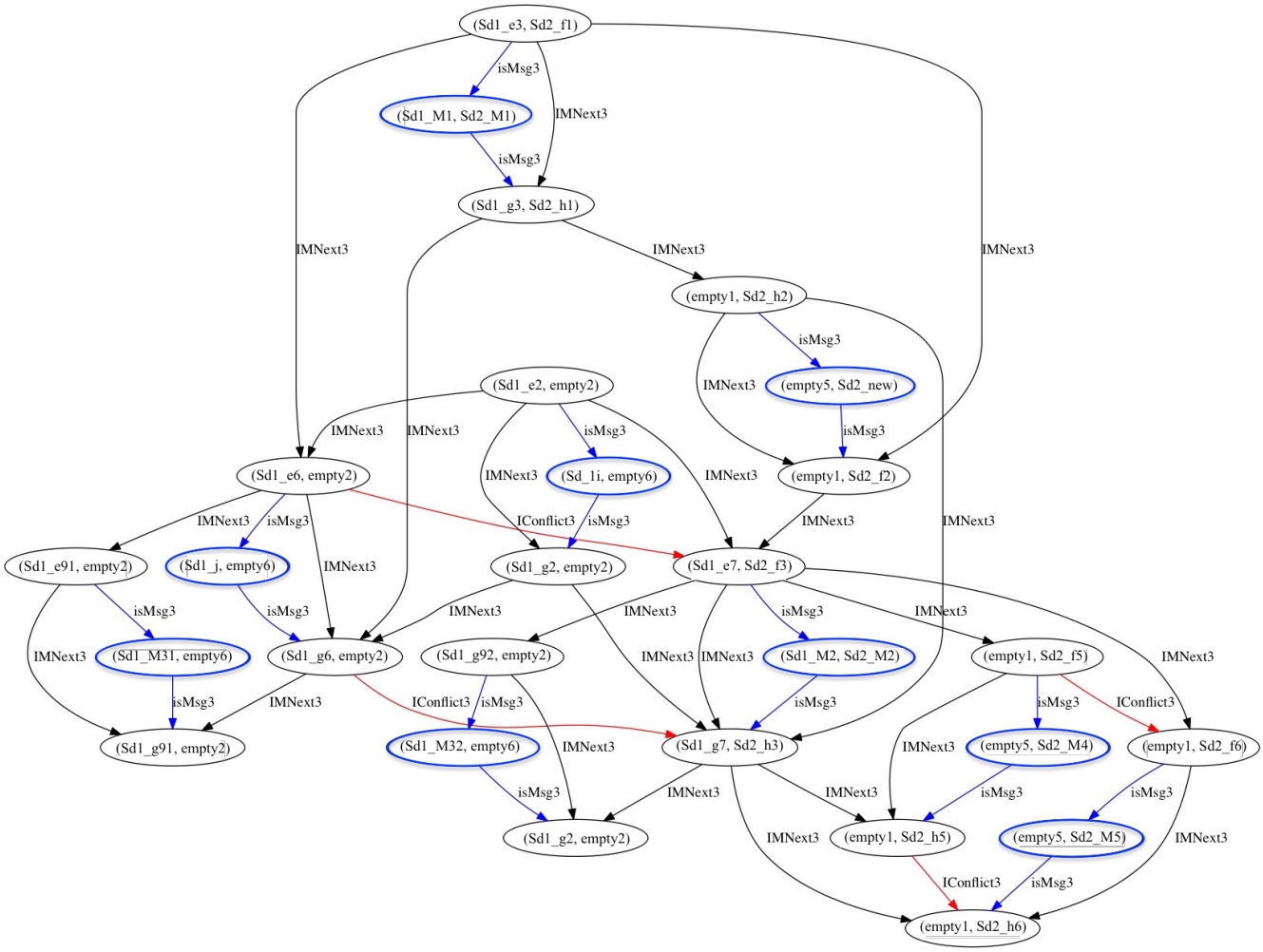


Figure 5.26: The composition results of diagrams sd1 and sd2 (Figure 2.5)

Figure 5.26 shows the graph representing the composition result of diagrams sd1 and sd2 (Figure 2.5). The graph depicts the sequence diagrams messages, send/receive events with the relations $iMNext_3$, $iConflict_3$, and $isMsg$. The graph shows the messages and $isMsg$ relations highlighted in blue, whereas the $iConflict_3$ relation is highlighted in red to simplify the analysis. The graph in Figure 5.26 shows that the messages `(Sd1_M1,Sd2_M1)`, which are composed occur in parallel with `sd1_i` as there is no *iMNext3* relation between their events. Moreover, these messages are followed by messages `Sd1_j` and `(sd1_M2, sd2_M2)`, which are in conflict relation as they are belonging to the *alt* CombinedFragment. `Sd2_new` is shown

in parallel with `Sd1_i` and `Sd1_j`, but it always comes after `(Sd1_M1,Sd2_M1)` as their events are connected via *iMNext* relations. Finally, `Sd1_M3`, which defined as `Sd_M31` and `Sd_M32` comes after the message `Sd1_j` and `(sd1_M2, sd2_M2)`, but `Sd1_M3` is parallel with messages `Sd2_M4` and `Sd2_M5`, which are in conflict, as they belong to the Combined-Fragment, *alt*. Figure 5.27 illustrates the Z3 solution in LES.

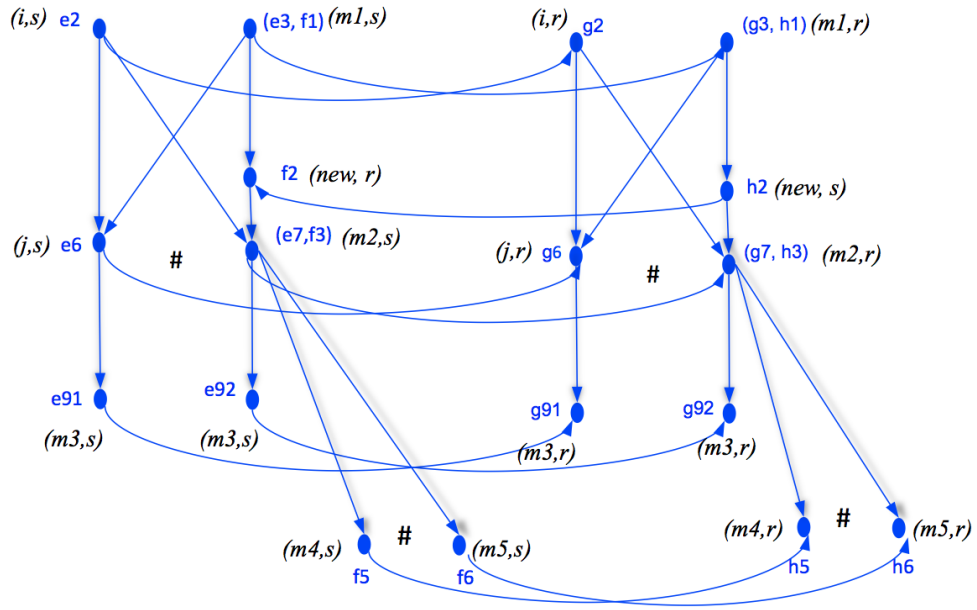


Figure 5.27: LES model for the Z3 solution in Figure 5.26

5.3.3 Preserving Semantics

In the model composition, it is essential that the composed model cannot provide other behaviour than what is described in the input models. Thus, the correctness of the composed model refers to the preservation of the semantics between the composed and input models. As a result, every trace in the graph of the composed model, which is referred to as G3 in Figure 5.1, if it is projected to the first coordinates, will appear in one of the input modes. This means that the graph G3, after eliminating the trace of execution (the events and the relations) that corresponds to Z3 model 2, is isomorphic with a sub-graph of G1 that represents the Z3 model 1. The same is true for G2.

As mentioned in section 6.2.7, there are different ways of proving graph and sub-graph isomorphism. However, proving sub-graph isomorphism mathematically, as mentioned earlier,

is out of the scope of this thesis. Instead, we test the sub-graph isomorphism using the *igraph* package that implements the VF2 algorithm [38].

To perform this check, we must first project the graph G3 trace of execution by eliminating the traces of Figure 5.26, which correspond to sequence diagram 2. Therefore, the only traces remaining will belong to sequence diagram 1, as Figure 5.28 shows. Secondly, these graphs are uploaded to the R studio tool, in order to check whether the graphs are isomorphic, using the command "graph.subisomorphic.vf2". In this case, the tool confirms that graphs A and B in Figure 6.23 are sub-graph isomorphic, as Figure 5.29 shows. The same procedure was performed for the Z3 graph that represents the LES' of sequence diagram 2. After eliminating traces of Figure 5.26 that correspond to the LES' of sequence diagram 1, the tool will confirm that the projected graph of the composed model is an isomorphic sub-graph.

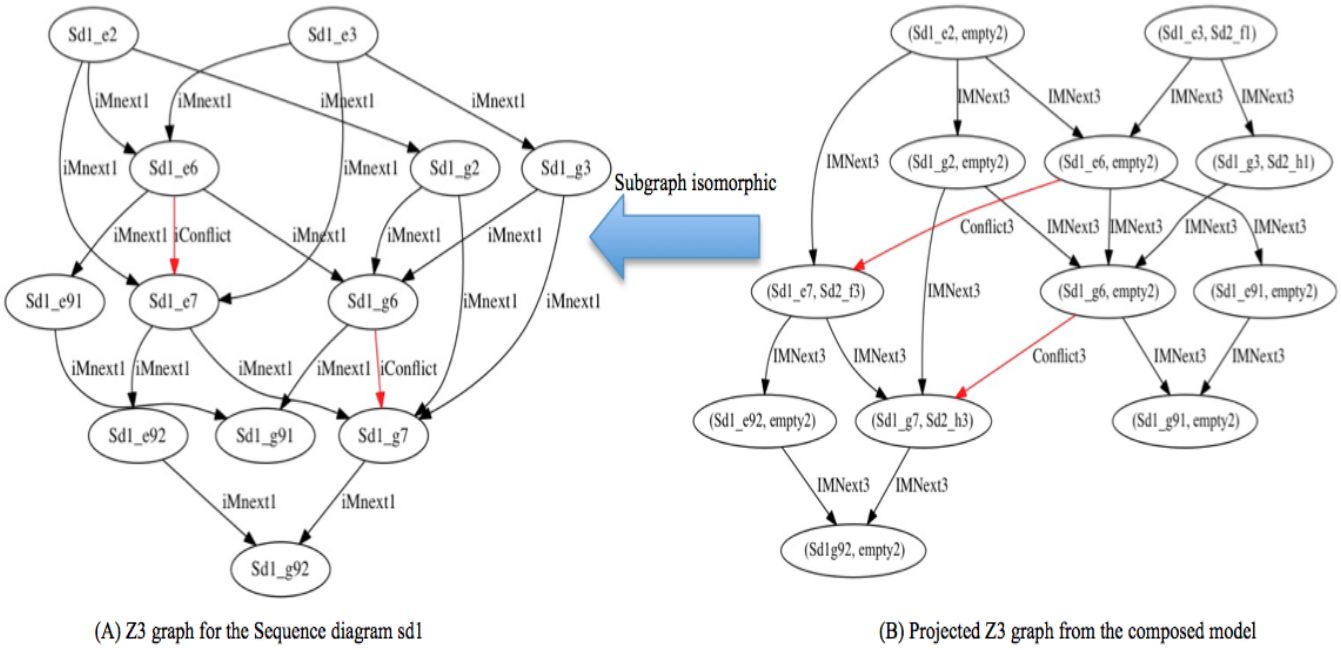


Figure 5.28: Z3 graph for sequence diagram sd1 and the projected Z3 graph from the composed model

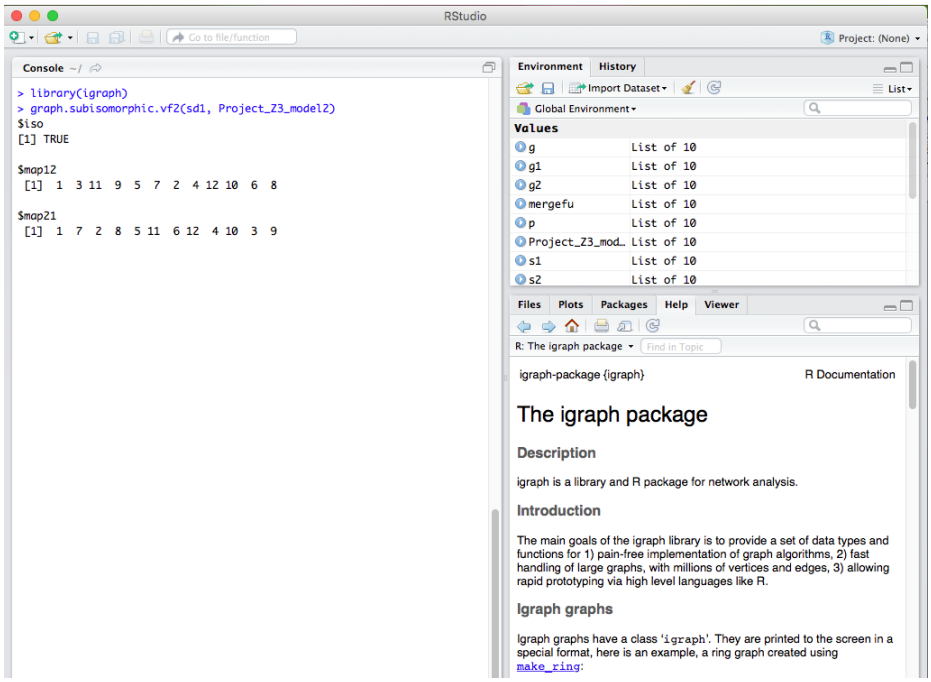


Figure 5.29: R studio proves the graph is sub-graph isomorphic between graphs (A) and (B) in Figure 5.28

5.4 Example

This section shows an example of automated aspect weaving via Z3. Aspect weaving is one of the model composition techniques and the aim of using it is to prove that this automated approach is fixable and can be applied to different kinds of composition. The example describes a petrol station scenario which was adapted from [68]. Let us consider the base model first as shown in Figure 5.30.

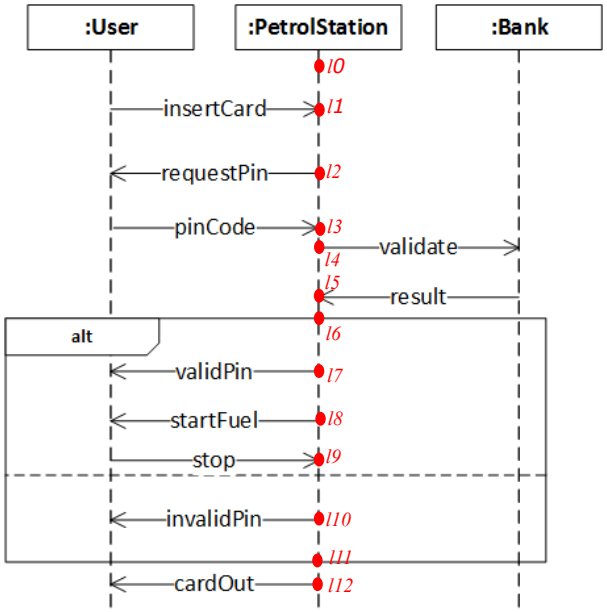


Figure 5.30: Petrol station base model

In this scenario, a user of a petrol station can only fill their car with petrol provided they have a card (and know the pin code for the card). The scenario starts with the user inserting a payment card $(insertCard)$. The petrol station requests the pin code from the user $(requestPin)$, which the user then enters $(pinCode)$. The petrol station sends a message to the bank to validate the pin code ($validate$ and $result$), and an alt fragment is used to model the two possible outcomes: (1) the pin code is valid. In this case the user is allowed to start fuelling $(startFuel)$ and when the user has finished he/she stops $(stop)$; (2) the pin code is invalid. In this case the user is informed that the pin code entered is invalid $(invalidPin)$. In both cases, the scenario ends by ejecting the card $(cardOut)$.

Now assume that we want a more refined model where we allow the user to indicate the exact amount of fuel required in advance. This is added by modelling an advice as shown in Figure 5.31.

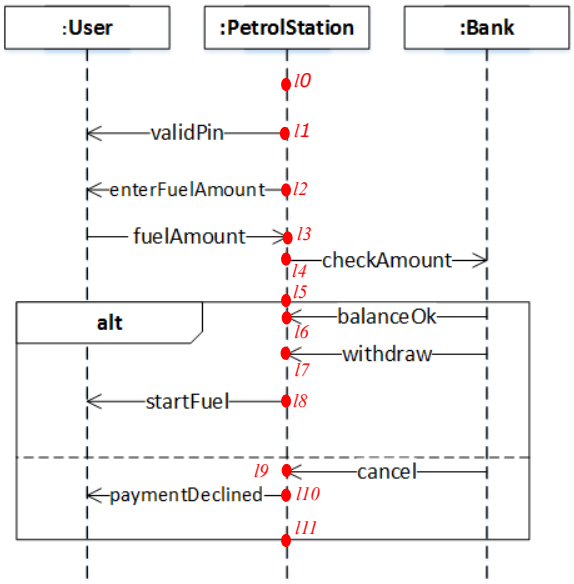
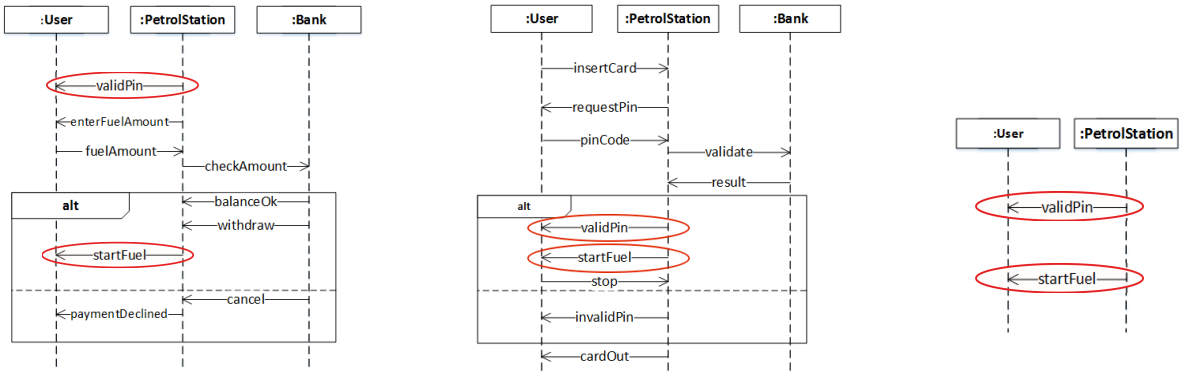


Figure 5.31: Petrol station advice model

The advice model starts with a valid pin code scenario. The idea here is that after entering the amount of fuel requested the petrol station forwards a message to the bank to validate whether the request is acceptable (basically the user has enough balance to cover the request). Again two options are possible. If the account balance covers this amount, it will be debited from the bank and the petrol station will start fuelling. However, if the account balance cannot cover the amount requested the transaction is cancelled.

To consider the advice within the original base model corresponds to weaving it into the base model and obtain an augmented model. Strictly speaking we can have more than one base model in a system and may want to integrate more than one advice. Without loss of generality we can assume that we can first obtain a composed model for the base behaviour and deal with weaving of an advice one at a time. More importantly, in order to perform the weaving itself, we specify a pointcut which shows exactly how the elements in the base and advice models match. The pointcut in Figure 5.32-(C) shows that the lifelines and messages $validPin$ and $StartFuel$ are matched (highlighted in red).



(a) Advice (b) Base (c) Pointcut

Figure 5.32: The pointcut mechanism

After producing the Z3 code for the advice and base by following the aforementioned transformation rule (see the appendix C), we need to create a Z3 model that links these them together based on the pointcut that defines where the advice should be inserted in the base model. This involves creating a set of constraints which identify which part of the base model must be matched to the advice. In that sense, the matching relates model elements of the base and advice together. The pointcut identifies $joinpoints$, that is, model elements which should be matched. There is a wide range of interpretations of how pointcuts should be used to match model elements of the base and advice. Wimmer et al. [155] survey some of these interpretations. To produce the Z3 code that glues the advice and base, any chosen interpretation selected must be formalised. For example, Klein et al. [90] introduce and formalise four interpretations. These four interpretations describe the degree of strictness when trying to detect a set of model elements which relating to another. For example in Figure 5.32, if we are looking for the message $validPin$ followed by $startFuel$ between two lifelines, we can be very strict and assume that the only acceptable match for this is to have the two messages appearing consecutively in a diagram. Alternatively, we can be less restrictive and allow a match provided every occurrence of message $validPin$ occurs before $startFuel$ irrespective of the behaviour that may occur in between the messages. Klein et al. [90] refer to the later as the $general$ interpretation, which our implementation follows. It is possible to replace this and follow any of the other three alternatives. However, for example choosing the strict interpretation will not allow weaving of the

models depicted in Figure 5.32.

In fact, the value of the match function can be obtained from the $pointcut$ model which describes which elements can be matched. The $pointcut$ shows that messages $validPin$ and $startFuel$ are matched in both models. Moreover, it can be observed also that the lifelines $Bank$ matched as they hold the same name which can be compose together even if non of their events are matched. The following snippet of Z3 shows the code for elements matching between the advice and base.

```
s.add (MessageMatch (sd1_validPin, sd2_validPin))
s.add (MessageMatch (sd1_startFule, sd2_startFule))
s.add (LifelineMatch (Base_Bank, Advice_Bank))
```

Once the messages are matched, send/receive events matched and thus their lifelines matched based on the rule of the glue. On the other hand, only lifelines $Bank$ are composed due to none of the events belong to them are matches.

Finally, it is possible that multiple instances of the advice messages to be found in the base. For example consider the scenario where validPin and startFuel appear twice or more in the base. In such cases, we would follow the Per Pointcut Match strategy introduced in [108] which assume that a new instance of the advice element is introduced for each pointcut match. For our example the model obtained corresponds to the diagram shown in Figure 5.33 which weaved the advice exactly as expected in the base model. The complete Z3 code for the petrol station example is presented in Appendix C.

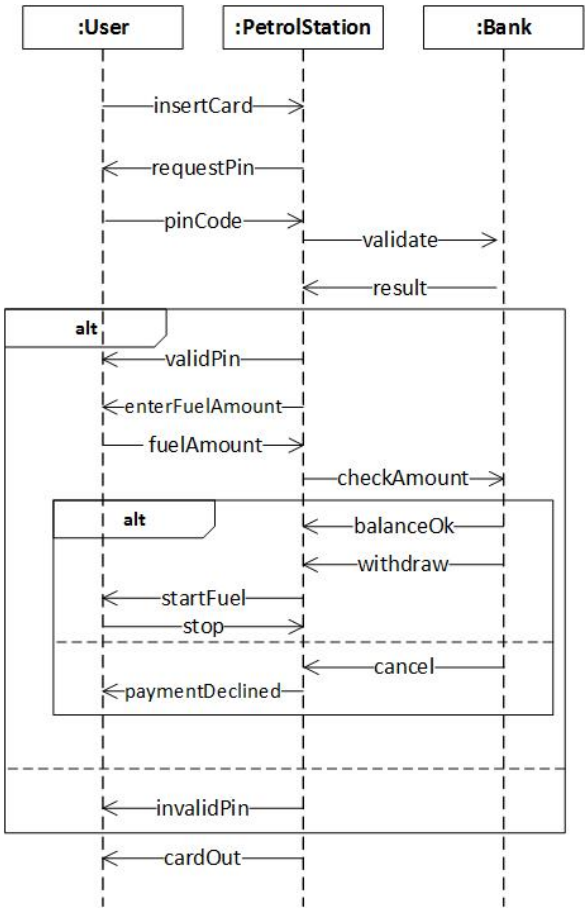


Figure 5.33: Woven sequence diagram

5.5 Limitations of the Approach

In this chapter, an automated method of sequence diagram composition via Z3 has been presented. The transformation illustrates the number of rules which transform part of the elements of a sequence diagram. However, this approach focuses more on the combining of some of the sequence diagram elements, such as lifelines, events, messages, and the LES operator; the operators representing the behaviour of the CombinedFragments, such as 'parallel' and 'alternative'. Therefore, there is no need to duplicate the representation of the CombinedFragments. As previously established, this work focuses on LES representation in Z3. However, LES does not support all operators of sequence diagrams, such as an 'option' or a 'loop' CombinedFragment, which are the main interest for future research in this case. Therefore, these operators are not covered in this approach.

As mentioned in Chapter 4, the option is semantically equivalent to an alternative CombinedFragment, but contains only one interactionOperand. The transformation of an option can be performed in the same way as for an alternative CombinedFragment, using a conflict operator. The condition attached to the CombinedFragment option can be defined as an axiom that is associated with the constant defining the CombinedFragment.

To represent the loop CombinedFragment, the LES and its equivalent Z3 model must model all possible iterations of the loop as 'unfoldings' (traces in the LES). As aforementioned, in constraint solvers, a finite number of possible iterations and hence 'unfoldings' must be assumed. This means modelling all possible iterations of the loop, with the transformation needing to define the number of constants equal to the number of iterations, in order to represent the messages belonging to the loop, as the graph shows below .

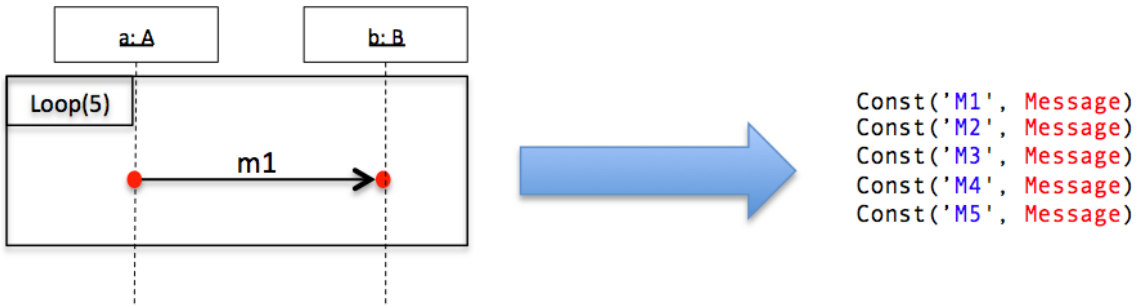


Figure 5.34: Finite loop in Z3

5.6 Chapter Summary

This chapter represents the automatic composition of sequence diagrams via Z3. The composition in this approach is carried out on LES and at the level of the sequence diagram, since both aspects of the models have been incorporated into the transformation algorithm, in order to generate a Z3 code. The transformation rules that map the elements of the sequence diagram and its semantics into Z3 syntax are discussed in section 6.2. The composition rules in Z3 have been explained in section 6.3. Finally, section 6.4 presents the example of a petrol station, whereby this approach was applied for the purposes of evaluating it and ensuring that there were no shortcomings in the performance of Alloy in Z3 and moreover, that the result of the composition was as expected. The next chapter presents a detailed comparative study of the composition of sequence diagrams via Alloy and Z3, in terms of performance.

CHAPTER 6

COMPARISON OF ALLOY AND Z3 FOR THE COMPOSITION OF SEQUENCE DIAGRAMS

6.1 Overview

In this chapter, a comparative study between Alloy and Z3 constraint solvers is presented from a performance perspective. Specifically, Alloy and Z3 are compared in terms of their composition of sequence diagrams, in order to evaluate the two methods described in Chapters 4 and 6. In addition, to compare the performance of Alloy and Z3, a number of sequence diagrams were composed using the logical constraints produced from the rules presented in Chapters 4 and 6 respectively.

6.2 Performance

In this section, the aim of the evaluation is to measure the performance of Z3 and Alloy constraint solvers, relative to the time required to compose sequence diagrams. This study has evaluated the composition time reported by Z3 and Alloy, as measured in this evaluation, in addition to the number of clauses produced in both constraint solvers.

In total, 14 experiments, divided into three phases, were carried out. In the first phase, the testing began with the use of sequence diagrams without CombinedFragments. Moreover, in the first experiment in Phase 1, two simple sequence diagrams were composed, each consisting of

four messages, as shown in Figure 6.1. In the following experiments, the number of messages was increased until the composition time became very prolonged.

In the second phase, one of the Phase 1 examples was selected and the number of lifelines increased to test how this change would affect the solvers performance.

Finally, the third phase used the same example as Phase 2 and inserted a CombinedFragment, in order to increase the complexity of the example. The nested CombinedFragments were then increased until the model became large and complex. The aim of this was to evaluate how this complex model would affect the speed of the solvers.

In this evaluation, the latest version of each constraint solver was used. The Z3 solver version was 4.4.1, while the version of Alloy Analyzer used was 4.2. The machine selected for this test had the following configuration: a MacBook Pro laptop running the Macintosh operating system on 2.5 GHz Intel Core i5, with 8GB RAM. Finally, the Alloy code of the experiments in this chapter was automatically generated via an SD2Alloy tool, whereas the Z3 code was generated manually.

6.2.1 Experiment Phase 1

In this phase, the approach was tested using sequence diagrams without CombinedFragments, as mentioned earlier. This phase consisted of eight experiments, starting with four messages, two lifelines and 12 events in each diagram, as shown in Figure 6.1.

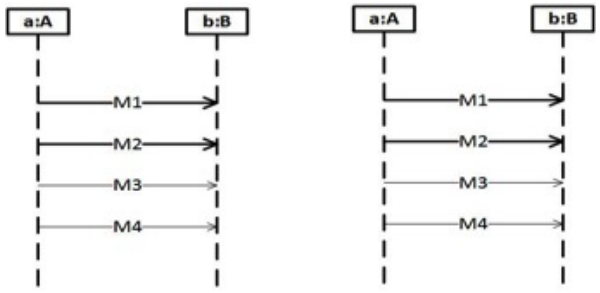


Figure 6.1: Sequence diagrams with four messages

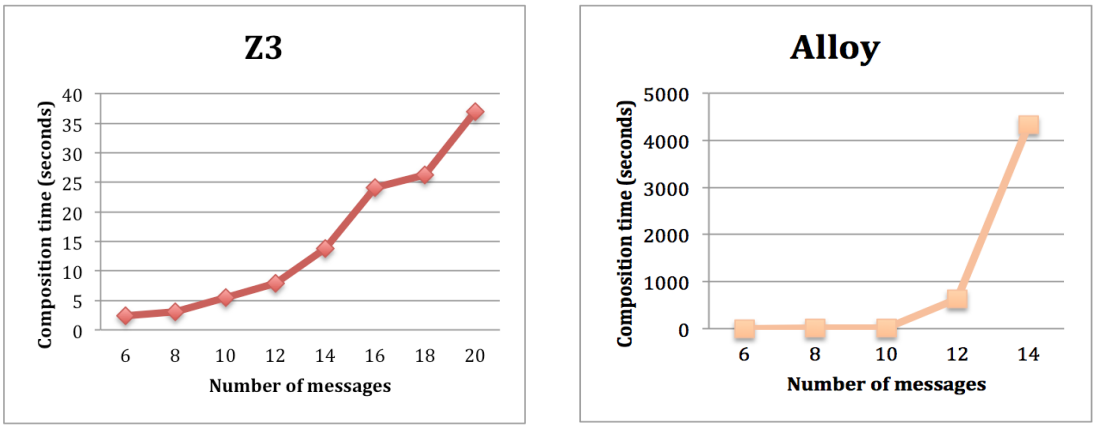
In this experiment, it was assumed that messages $M1$ and $M2$ were matched in both diagrams. This means that their events (send/receive) and their lifelines were also matched in both diagrams.

Table 6.1: Phase 1 experiments

Example	Total	Lifelines	Messages	Events	Alloy		Z3	
					Time (sec)	Clauses	Time (sec)	Clauses
1	22	2	6	12	0.31	210603	2.38	7590
2	26	2	8	16	5.91	373819	3.05	12853
3	32	2	10	20	9.02	609675	5.51	36805
4	38	2	12	24	630.12	818146	7.87	52542
5	44	2	14	28	4329.79	1344924	13.73	220154
6	50	2	16	32	Timeout	1870923	24.10	300227
7	56	2	18	36	Timeout	2505690	26.21	464612
8	62	2	20	40	Timeout	3294658	36.95	582375

Table 6.1 shows the Phase 1 experiments in detail. For example, the example 1 shows the results of the first experiment. The total number of elements in the final model, resulting from the composition, was 22 (the overall elements of the composed sequence diagram). The composition time shows that Alloy (0.31 seconds) was faster than Z3 (2.38 seconds). This experiment illustrates that Alloy had a shorter composition time than Z3. However, the following experiment shows that increasing the number of sequential messages strongly affected the performance of Alloy. Overall, this study shows that approximately one hour and 20 minutes is required to compose sequence diagrams containing 14 messages. Moreover, this model, which contains 14 messages, has 1,344,924 clauses, as shown in Table 6.1. However, increasing the number of messages increased the number of the variables and clauses in the model, which made the solver run out of memory when solving the model, as experiments 6-8 show in Table 6.1.

On the other hand, Z3 showed good performance throughout most of the experiment. Increasing the number of messages did not have a significant effect on its performance, which was less than one minute on average, as shown in Figure 6.2-a.



(a) Composition time in Z3

(b) Composition time in Alloy

Figure 6.2: Composition time in Z3 and Alloy

6.2.2 Experiment Phase 2

In this phase, the approach used in this study was evaluated by increasing the number of lifelines to determine how the change would affect the solvers performance. Moreover, also in this phase, one of the Phase 1 examples (example 5) was adopted as a test case. This already had a performance issue, as shown in Table 6.1. The number of lifelines was then increased to the point at which a significant change in performance was evident. In this phase, three experiments were conducted, starting from three lifelines in each diagram in the first experiment. The number of lifelines was then increased until the performance showed a dramatic change.

Table 6.2: Phase 2 experiments

Example	Total	Lifelines	Messages	Events	Alloy		Z3	
					Time (sec)	Clauses	Time (sec)	Clauses
9	45	3	14	28	10020.620	1609394	14.67	235980
10	46	4	14	28	10603.436	1719922	15.08	280193
11	47	5	14	28	Timeout	1837535	20.33	342690

Table 6.2 shows the results of the Phase 2 experiments. This study confirms the earlier findings for Phase 1, namely that Alloy's performance was strongly affected by the number of

elements in the model. This study therefore confirms the Phase 1 results by demonstrating that when the number of clauses in the composed model exceeds 1,800,000 clauses, Alloy Analyzer will run out of memory, as shown in Tables 6.1 and 6.2.

Z3 still performs well and consistently in the above circumstances. In this study, increasing the number of lifelines did not have a significant effect on Z3's performance.

6.2.3 Experiment Phase 3

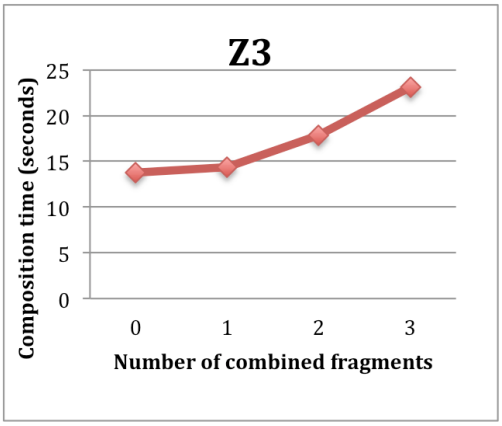
In Phase 3, the experiments tested how CombinedFragments affected the performance of Alloy and Z3. Again, in this phase, example 5 was adopted as a test case and a CombinedFragment was inserted. The number of nested CombinedFragments was then increased until one of the solvers ran out of memory. Table 6.3 shows that when messages were further structured through CombinedFragments, Alloys performance was strongly affected and the solvers speed was also slowed down (Figure 6.3-(b)).

Table 6.3: Phase 3 experiments

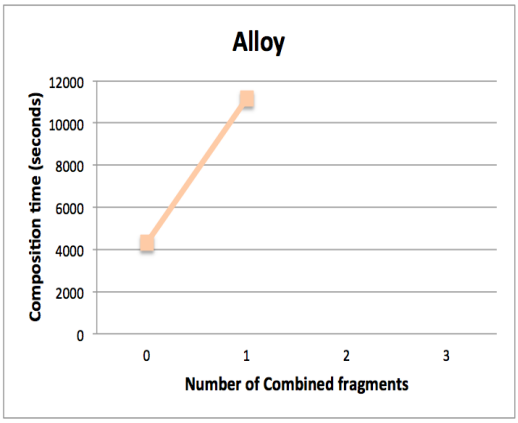
Example	Total	Combined Fragment	Lifelines	Messages	Events	Alloy		Z3	
						Time (sec)	Clauses	Time (sec)	Clauses
12	47	1	2	14	30	11163.872	1753293	14.32	285163
13	48	2	2	14	30	Timeout	2281797	17.85	393111
14	49	3	2	14	30	Timeout	2348862	23.08	409395

This study also confirms that Alloy's performance was affected by the number of clauses, as mentioned earlier. Indeed, with an increasing number of CombinedFragments, the performance of Alloy becomes very slow. In example 10, Alloy's composition time was about three hours. In examples 11 and 12, Alloy ran out of memory, which shows that the maximum number of clauses Alloy can solve is 1,753,293.

Z3, on the other hand, showed steady performance and increasing the CombinedFragments did not have a significant effect on its performance.



(a) Composition time in Z3



(b) Composition time in Alloy

Figure 6.3: Composition time in Z3 and Alloy

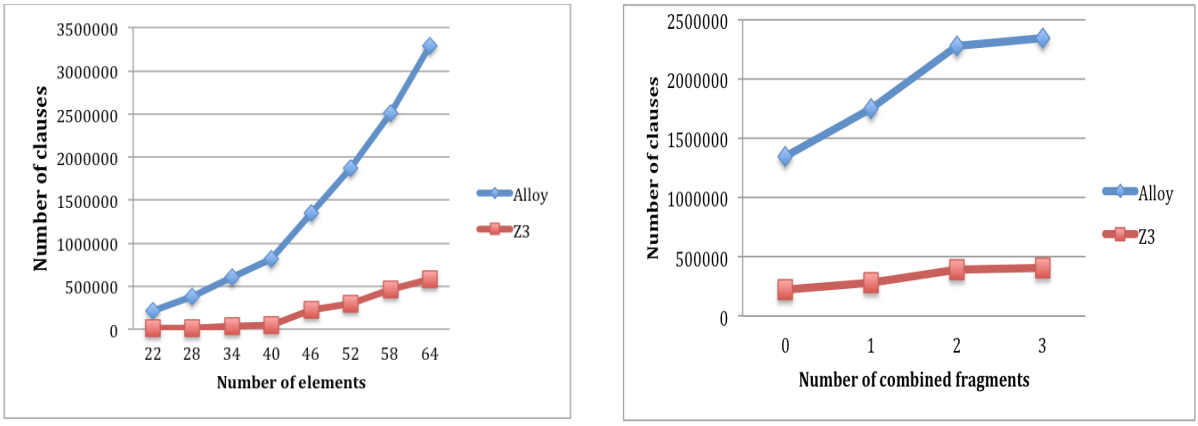
6.2.4 Discussion

This chapter has described a comparison study that evaluated the constraint solvers from a performance perspective. This study was divided into three phases. Each phase evaluated the performance of the two constraint solvers when increasing a major element of the sequence diagrams. For example, Phase 1 showed the effect of increasing the messages in the sequence diagrams on the constraint solvers. The second phase evaluated the performance of both solvers when increasing the number of lifelines. Finally, the third phase tested how CombinedFragments affect the performance of Alloy and Z3.

In this study, Z3 demonstrated good performance throughout most of the experiments in all phases and increasing the number of elements did not have a significant effect on this performance (less than one minute on average).

After closer inspection, the scalability problems in Alloy seemed to be due to the fact that Alloy Analyzer, which underlies Alloy, is based on a SAT solver. SAT-solving time is known to vary enormously, depending on factors such as the number of variables, the ordering of clauses and the average length of the clause [46].

In terms of Z3, there are several reasons why it performed better than Alloy. Firstly, Z3 uses many heuristics to eliminate quantifiers in formulae. It uses an e-graph to instantiate quantified



(a) phase 1.

(b) phase3.

Figure 6.4: Number of clauses in Z3 and Alloy for Phase 1 and 3.

variables, code trees, and eager instantiation, which makes it very effective at dealing with quantifiers [41, 114].

The second reason is that the implementation languages are different in Z3 and Alloy. For example, Z3 was implemented in C++, while Alloy and its SAT-solver were implemented in Java. Another reason that might make Z3 more efficient is that SMT solvers operate at a higher level of abstraction than SAT solvers. SMT solvers can use information about the *structure* and *semantics* of a formula to speed up the satisfiability process, whereas a SAT-based approach converts the model to SAT formulae using Boolean encoding [114]. Due to the increasing size of the Boolean encoding, an exponential increase in composition time then occurs. It was observed that the size of Z3-SMT clauses is much smaller than what is produced by Alloy, which uses a SAT4J solver (Figure 6.4).

6.2.5 Chapter Summary

This chapter has presented a comparative study between Alloy and Z3 constraint solvers from a performance perspective. This comparison study aimed to evaluate methods of using Alloy and Z3 constraint solvers for composing sequence diagrams. This study showed that Z3 was much faster than Alloy in most of the experiments. As a result, several questions that merit further investigation have arisen. For example, further investigation is required to determine the precise

reason for the differences in performance between the two solvers.

CHAPTER 7

CONCLUSION AND FUTURE WORK

This chapter concludes the work presented in this thesis. In section 8.1, the contributions made in this thesis are summarised. Section 8.2 outlines a discussion on any future work, which could be carried out to expand and improve this research.

7.1 Summary of Contributions

The main contribution of this thesis is the presentation of two automated methods of sequence diagram composition, using the constraint solvers, Alloy and Z3. The outline of the Alloy composition method involves the creation of two Alloy models. Each model created consists of sets of logical constraints, uniquely identifying each component of their corresponding sequence diagram by restricting the metamodel. To combine the models, additional constraints capturing the composition glue were produced. This glue specified which elements needed to be composed, along with where the elements should be inserted, and the ways in which the composition process worked to obtain the expected result.

To ensure the correctness of the composition process, the semantics of the composition were formalised with the help of Labelled Event Structures (LES). The result obtained automatically with a constraint solver was preserved in the formal interpretation of the present composition. The Alloy-based automated method of composition was implemented here as an Eclipse plugin called *SD2Alloy* to compose the sequence diagrams. The evolution of the *SD2Alloy* then revealed a performance issue in Alloy, occurring in the composition of large models. In order to

counteract it, an alternative contribution to automated sequence diagram composition using the Z3-SMT solver was presented in this study. In addition, the other advantage of using Z3 was that it is capable of displaying the overall model in a single solution; whereas Alloy produces as many solutions as there are possible traces in the model, with each solution representing a different trace. Therefore, Z3 provides engineers with a better solution for assisting in understanding overall behaviour.

In the above approach, a number of transformation rules were defined to map the elements of the sequence diagrams and LES metamodels to these Z3 metamodels. Using this method, each sequence diagram and its eliminated version of LES' were automatically transformed into Z3; these being instances of the Z3 metamodel. Here, this transformation produced sets of Z3 logical constraints that uniquely identified each component of their corresponding sequence diagram and LES' model. Solving this Z3 model will produce only one solution, which is isomorphic with LES' model of the sequence diagram.

Finally, in order to compose the Z3 models representing the input sequence diagrams, the set of logical constraints representing the composition glue was added; matching the common elements of the input models. These logical constraints consisted of a number of axioms that were able to match all possible scenarios that covered by this approach. Similar to Alloy, Z3 was used in this study to formally check whether the sequence diagrams could be composed and to automatically compose the diagrams. We believe that the methodology used in this thesis to automate the composition sequence diagrams could be generalised and applied in various composition domains. Therefore, an aspects-oriented case study was applied and woven via the Z3-SMT solver.

This approach should be applicable to a wide range of modelling notations used for design. Although the composition of sequence diagrams has been the focus in this instance, other kinds of model can also be composed, e.g. class diagrams, communication diagrams and Message Sequence Charts (MSC). Finally, this thesis presents a comparison study between Alloy and Z3 from the point of view of performance.

Chapter 2 began with an overview of some of the basic concepts related to UML modelling,

interaction semantics, model composition and the technologies used to support composition; in particular, constraint solvers, such as Alloy and Z3. This was followed by a review that explores current approaches to composition via constraint solvers. The review presented a number of different constraint solvers used to compose models, as well as the challenges, benefits and trade-off of needs to be considered when composing a model. From the background, it may be gathered that current approaches use manual composition or algorithms to compose behavioural models. Furthermore, most methods using algorithms are designed to compose simple sequence diagrams, without the CombinedFragments that represent complex behaviour.

The objective of the background provided was to map out the main activities used, in support of the composition of dynamic models, as well as identifying the gaps in current approaches. From the background, it became apparent that the approaches reviewed do not fully cover the automated composition of dynamic models. In Chapter 3, the methodology used for model composition was demonstrated. In particular, a technique was presented in this study, called Exact Metamodel Restrictions (EMR). This described the mapping between dynamic models into logical constraints. This was followed by composition semantics, which guide the composition to produce the expected results. In addition, the syntactic and behaviour glue used for model composition was illustrated.

In Chapter 4, sequence diagram composition via Alloy was described. This involved sets of transformation rules that map the sequence diagram elements to Alloy. Logical statements of Alloy were produced through EMR. In addition, this chapter demonstrated the composition of sequence diagrams via Alloy. It involved algorithms demonstrating the process of generating logical statements to represent the composition glue.

In Chapter 6, an alternative composition approach was revealed using Z3. The aim of this approach was to resolve Alloy's poor performance and use the advantage of Z3's ability to represent the overall model in one solution. This Chapter described the process of composition; carried out in this study on the level of both the sequence diagram and LES. Further to the above, the Chapter consisted of three main sections. The first section demonstrated the mapping between the sequence diagram and LES to Z3, while the second section demonstrated the

composition mechanism. Finally, the third section showed the evaluation of the approach, using an aspect-oriented case study.

Chapter 7 then presented a comparison study between Alloy and Z3 from the performance point of view. This comparison study was based on running 14 experiments, with the above Chapter confirming that Z3 performs better than Alloy in most of the evaluation experiments; especially for the composition of complex sequence diagrams. In this Chapter, Alloy's performance issues were investigated and a number of the reasons underlying such problems were presented.

7.2 Future Work

Following the advances made in this thesis, a number of directions for future research have arisen. Some of these extensions could help overcome a few of the limitations of this research, whilst others could provide additional capabilities.

The sequence diagram metamodel used in this research, as presented in Figure 2.6, is a subset of the UML metamodel derived from [116]. However, there are certain elements existing in the UML metamodel for sequence diagrams that have not been included in the metamodel used in this research; such as loop CombinedFragment. As seen in [54], LES offer suitable semantics for sequence diagrams and the various interactive fragments defined; whereas operators, such as *seq, alt* and *par* have a natural correspondence to relations within LES and it may be less obvious how to capture other operators. To represent a loop fragment, the LES must model all possible iterations of the loop as 'unfoldings' (traces in the LES) as explained in Chapters 4 and 6. Moreover, in constraint solvers, a finite number of possible iterations must be assumed. Thus, the representation of the loop will be revealed as a limitation of the current approach in terms of how to present an infinite number in the constraint solver. This remains a task for future work.

The CombinedFragment option could also be considered for future work. The option is semantically equivalent to an alternative CombinedFragment, but contains only one Interaction-

Operand. The transformation of an option can be performed in the same way as the alternative CombinedFragment, but only one operand will be generated for it instead of two, as stated in the definition. The occurrence is known to be associated with the condition. This condition can be encoded as a fact in Alloy, or as function and axiom in Z3.

Other future work would consist of composing state machines via constraint solvers, such as Z3. The transformation from state machines to Alloy has been presented in several approaches [59, 150]. However, performing the composition via Alloy is not a suitable choice, as mentioned in Chapter 7, as it reveals Alloy's performance issues. Instead, it is currently planned to transform state machines to Z3. This transformation is based on the interpretation of state machines in Alloys logical statement. Thus, translating these logical statements to Z3 can save time and will guarantee that the transformations are correct and have been evaluated. In terms of the composition, it necessarily involves studying all possible cases of state machines composition and improving the current glue to cover these cases, similar to what has been done with sequence diagrams.

In addition, other future work would involve enhancing the sequence diagram metamodel used in this approach to model transformation, so that it includes OCL constraints. OCL is a text-based language that uses first-order logic statements to provide constraints of the model elements in UML. As these are first-order logic statements, such constraints can be translated following the work presented in [8]. However the work proposed in [8] is designed for class diagrams. In this work, the sequence diagrams are targeted. For example, the pre- and post-conditions and the 'if' statement ('if then else') in the OCL statements can be written as a fact, where the 'if' statement can be translated to imply the operator. More specifically, the Alloy syntax for 'if-then-else' expressions is:

condition $\implies$ {expr1}
else {expr2}

Finally, plans are also being drawn up to improve the current composition methods, in sup-

port of a bi-directional model transformation between sequence diagrams and Z3, where composition is performed in Z3 and the results are transformed back into sequence diagrams. However this could be complicated, since Z3 language is more expressive than that of the sequence diagrams.

Appendices

APPENDIX A

SD2ALLOY: IMPLEMENTATION OF A COMPOSITION FRAMEWORK

A.1 Overview

This chapter will introduce the implementation of the transformation rules presented in the previous chapter. The approach will involve using a plug-in called *SD2Alloy*.

A.2 SD2Alloy Architecture

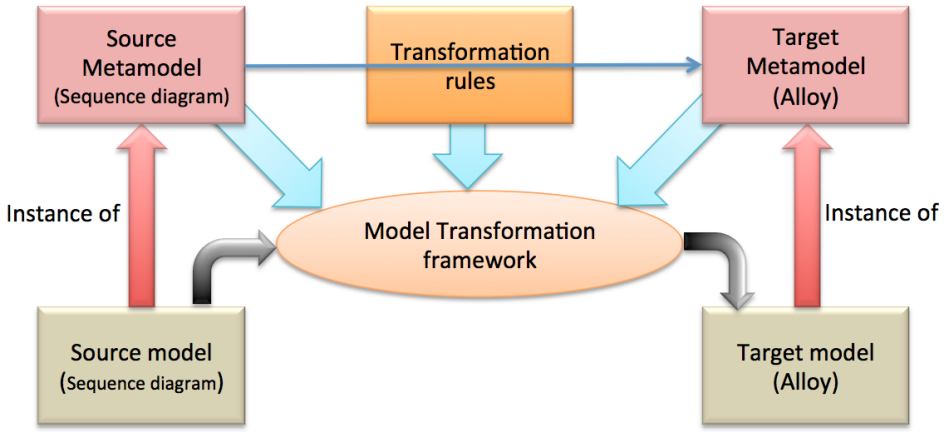


Figure A.1: Overview.

Figure A.1 presents an overview of the approach as explained in Chapter 4. In particular, the transformation rules have been defined to conduct the model transformation. The transformation rules map the elements of the sequence diagram metamodel onto the Alloy metamodel,

as Figure A.1 shows. Subsequently, these rules are executed via the Simple Transformer (SiTra) transformation engine. This means that every model arising from the source metamodel can be transformed automatically into an instance of the destination metamodel. Finally, the model transformation is implemented as an Eclipse plug-in application called *SD2Alloy*. Figure A.2 depicts the *SD2Alloy* architecture. The tool includes a modified open source tool called Papyrus [99], which allows the user to generate any number of sequence diagrams and exports these as XMI files, so they can be parsed. *SD2Alloy* parses the XMI files generated by Papyrus into Java objects, using the UML2 library [121]. SiTra is then used to transform the Java objects of sequence diagrams and create the Alloy Java object that will produce the Alloy code. Finally, the generated Alloy model can be analysed using Alloy Analyser.

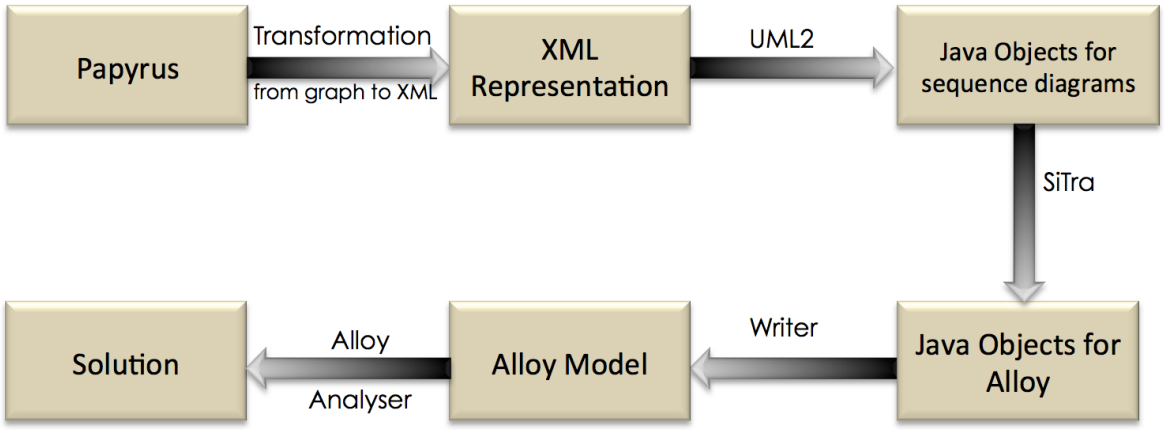


Figure A.2: Technologies used during the development of SD2Alloy.

A.3 Integration of Papyrus

The decision to integrate a Papyrus tool into *SD2Alloy* was based on the fact that it is a state-of-the-art UML open source tool with the power to support most of the sequence diagrams components, such as the combined fragments component (alt, par,loop, etc). Currently, several UML tools support sequence diagram such as ArgoUML [120], Poseidon [1], UMlet [14]. However, some of these tools are not open source such as Poseidon and other does not support all sequence diagram components such as CombinedFragments such as ArgoUML and UMlet.

Papyrus supports several UML diagrams, e.g. the class diagram, object diagram, state machine diagram, etc. However, in the *SD2Alloy* tool, the only component of UML models needed is a sequence diagram. Thus, it is necessary to integrate only the sequence diagram. As mentioned, Papyrus consists of a mix of different diagrams together; therefore it is difficult to separate them. To solve this problem, all diagrams have been deactivated and only keep a sequence diagram active.

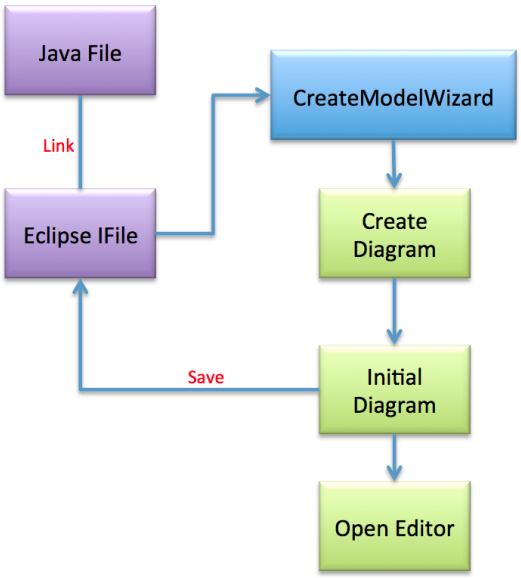


Figure A.3: Creation of sequence diagram.

Figure A.3 illustrate the process of generating sequence diagram in *SD2Alloy*. As Eclipse plug-ins (including Papyrus) can only control files inside its workspace, thus a link has been created between the files created in Java and Eclipse IFile which aimed to read the files outside the workspace. After that, a class called CreateModelWizard in Papyrus is used to initiate a sequence diagram with the files created. The Editor can then edit the sequence diagram, which is also integrated from Papyrus.

A.4 Generating an XMI for Sequence Diagrams

XML Metadata Interchange (XMI) [117] is a standard created by the Object Management Group to allow the interchange of metadata information. XMI is commonly used to express

UML models and as such, represents a widely accepted form of output in UML tools. UML tools allow UML models designed within the tool to be exported as XMI files. An example of a small snippet of XMI that represents a sequence siagram created using Papyrus is shown below.

```
<?xml version="1.0" encoding="UTF-8"?>
<uml:Model xmi:version="20110701" xmlns:xmi="http://www.omg.org/spec/XMI/20110701" xmlns:uml="
    http://www.eclipse.org/uml2/4.0.0/UML" xmi:id="_oOxawLQEEeOhB5bNYPu52A" name="model">
<packagedElement xmi:type="uml:Interaction" xmi:id="_oPTmQLQEEeOhB5bNYPu52A" name="
    Interaction1">
//CombinedFragment in XML type of par
<fragment xmi:type="uml:CombinedFragment" xmi:id="_VjJxoLTfEeOWZqCjYfMtHg" name="
    CombinedFragment1" covered="_r3cKkLQEEeOhB5bNYPu52A _q8BEILQEEeOhB5bNYPu52A"
    interactionOperator="par">
//First interactionOperand in the CombinedFragment1
<operand xmi:id="_VjKYsLTfEeOWZqCjYfMtHg" name="Operand1" covered="_r3cKkLQEEeOhB5bNYPu52A
    _q8BEILQEEeOhB5bNYPu52A">
<guard xmi:id="_VjKYsbTfEeOWZqCjYfMtHg">
<specification xmi:type="uml:LiteralString" xmi:id="_VjKYsrTfEeOWZqCjYfMtHg" value="undefined"
    />
<maxint xmi:type="uml:LiteralInteger" xmi:id="_VjK_wbTfEeOWZqCjYfMtHg" value="1"/>
<minint xmi:type="uml:LiteralInteger" xmi:id="_VjK_wLTfEeOWZqCjYfMtHg"/>
</guard>
//The events covered by Operand1 which called(e1,g1)
<fragment xmi:type="uml:MessageOccurrenceSpecification" xmi:id="_trZbgbQEEeOhB5bNYPu52A" name=
    "g1" covered="_r3cKkLQEEeOhB5bNYPu52A" message="_trYNYLQEEeOhB5bNYPu52A"/>
<fragment xmi:type="uml:MessageOccurrenceSpecification" xmi:id="_trZbgLQEEeOhB5bNYPu52A" name=
    "e1" covered="_q8BEILQEEeOhB5bNYPu52A" message="_trYNYLQEEeOhB5bNYPu52A"/>
</operand>
```

The XMI shown above represents just a small fragment of code that forms the entire sequence diagram. As can be seen in the XMI code above, the first part represents the CombinedFragment in the sequence diagram which called $CombinedFragment1$. The interactionOperator of the CombinedFragment defined as "par". This fragment contains an interactionOperand "$Operand1$". This operand as shown covered two events $(e1, g1)$. In fact, the code as shown is incomprehensible to most developers and decoding the XMI to obtain the sequence diagram information is a tedious process. However, this process could be done automatically using XMI

parsers.

A.5 Parsing XML Data into Java Objects

Parsing is a process of syntactical analysis and interpretation of a structured text [6]. As mentioned previously, Papyrus presents diagrams as XML files. As such, before the transformation to Alloy, parsing is required to interpret the XMI code generated using UML tools into Java objects that can be manipulated by SiTra. The parsing and generation of Java objects in the diagrams is performed using the UML2 library, as the code set out below shows.

```
// read XML file and return UML Objects
public class Xml2obj {

public static Model load(String filePath){
// init
ResourceSet resourceSet = new ResourceSetImpl();
org.eclipse.uml2.uml.resources.util.UMLResourcesUtil.init(resourceSet);
Model epo2Model = null;
// load from file
URI filrUri = URI.createFileURI(filePath);
Resource resource = resourceSet.createResource(filrUri);
resource.load(null);
org.eclipse.uml2.uml.Package package_ = (org.eclipse.uml2.uml.Package) resource.getContents().
    get(0);
epo2Model = package_.getModel();
return epo2Model;
}
}
```

Using the above method, a UML2 library automatically parsing XML files that Papyrus generated which includes all the information of the diagram and generate Java objects that corresponds to the original sequence diagram.

A.6 SiTra for Executing the Transformation Rules

The step following the process of parsing the XML files into Java objects is where the actual model transformation process takes place. This process is conducted using SiTra, which is a Java library that can provide a lightweight framework for performing transformations. SiTra has recently become a common choice for executing transformation rules, due to its usability [61]. It is also applicable to the conducting of large and complex transformations. As explained in Chapter 2, Section 2.8, SiTra contains two interfaces: the rule and transformer interfaces. The rule interface should be implemented for each transformation rule, whereas the transformer interface provides a framework for methods that carry out transformations.

The rule interface includes three main methods: $check()$, $build()$ and $setProperties()$. $check()$ method returns a Boolean value signifying whether this rule is applicable to the source object. The $build()$ method generates the target model element. Finally, $setProperties()$ is used to set the attributes and links for the newly created target element.

Since the rules for UML objects are similar, this section will present just one of them, namely a sequence diagram, Lifeline. For other UML objects, the source code should be referred to. The Lifeline2Alloy rule implements the Rule interface in SiTra. As mentioned above, the three rule interface methods require implementation ($check, buildandsetProperties$).

```
public class Lifeline2Alloy implements Rule{
@Override
public boolean check(Object source) {
if(source instanceof LifelineImpl){
return true;
}
else return false;
}
```

The above method shows the check method that should return a boolean value. Next, the built method returns the target object created by the information from the source element as the code below shows. A $HashMap$, as presented below, is used to store all the created objects.

```
public Object build(Object source, Transformer t) {
Lifeline lifeline = (Lifeline) source;

// add abstract for lifeline
// abstract sig LIFELINE {}
ASig lifelineAbstract = getSig("LIFELINE");
lifelineAbstract.set_attr(AAttr.ABSTRACT).;
lifelineSig.AddField("CLASS", lifelineClass).AddField("NAME", lifelineName);

// add the lifeline
ASig lifelineSig = getSig(lifeline.getName);
lifelineSig.set_attr(AAttr.ONE).set_parent(lifelineAbstract);
lifelineSig.AddField("CLASS", lifelineClass).AddField("NAME", lifelineName);
}
```

The above method shows that for all lifelines in the sequence diagram, the transformation generate an abstract signature which called "LIFELINE". Then for each lifeline, the transformation generate a keyword "ONE" followed by the signature name. Finally, the lifeline signatures fields will be added which consist of the lifeline name and class.

The rules then must be added into the transformer as the snippet of code illustrated below shows. A method called transformAll can then be invoked to automatically transform the UML diagram.

```
// List all of all classes that extend the rule interface
List<Class<? extends Rule<?, ?>>> rules = new ArrayList<Class<? extends Rule<?, ?>>>();
// Add rules to the list of rules
rules.add((Class<? extends Rule<?, ?>>) InteractionOperand2Alloy.class);
rules.add((Class<? extends Rule<?, ?>>) CombinedFragment2Alloy.class);
rules.add((Class<? extends Rule<?, ?>>) Interaction2Alloy.class);
rules.add((Class<? extends Rule<?, ?>>) Lifeline2Alloy.class);
rules.add((Class<? extends Rule<?, ?>>) Message2Alloy.class);

// Create the transformer
Transformer trans = new SimpleTransformerImpl(rules);

// Transform to alloy model
trans.transform(model.getOwnedElements());
```

A.7 SD2Alloy: An Eclipse Plug-in

The model transformation framework described in the previous chapters was implemented as an Eclipse plug-in called *SD2Alloy*. Figures A.4 and A.5 show two snapshots of the *SD2Alloy* interface. In both cases, the left panel shows a list of the current sequence diagrams used (here, $sd1.di$ and $sd2.di$, where di is the extension name given by Papyrus), as well as the syntactic matching declarations of model elements from the different diagrams (here, $sd1 - sd2Equality.eq$). Different levels of detail can be shown on different panes in the tool with the Editor in the middle, indicating the current diagram or code being edited. For example, in Figure A.4, the tool shows a diagram and in Figure A.5, it shows the Alloy code generated for the same diagram. Properties of elements being edited can also be seen and changed on a separate pane at the bottom right of the tool.

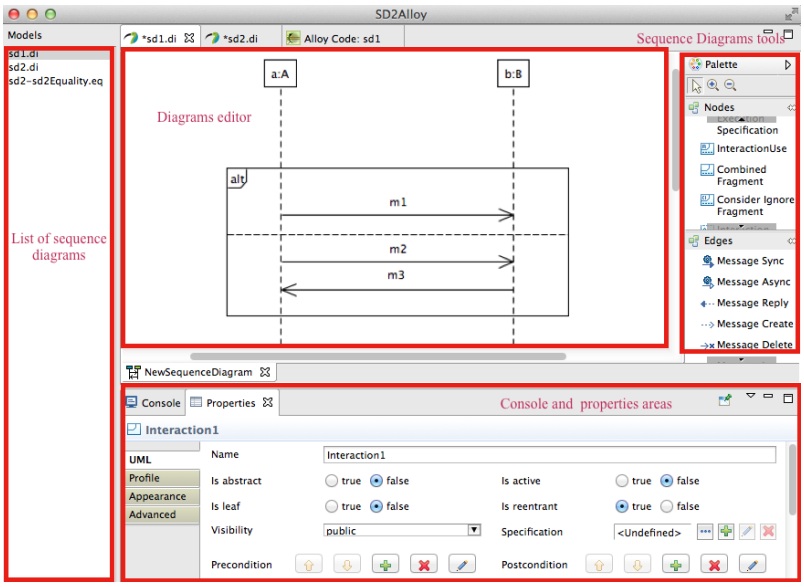


Figure A.4: A snapshot of the SD2Alloy interface.

A.8 Generating an Alloy Model from a Running Example

In order to use *SD2Alloy* to auto-generate Alloy from the sequence diagram, the *SD2Alloy* user should first provide those diagrams for the tool that need to be composed by drawing or

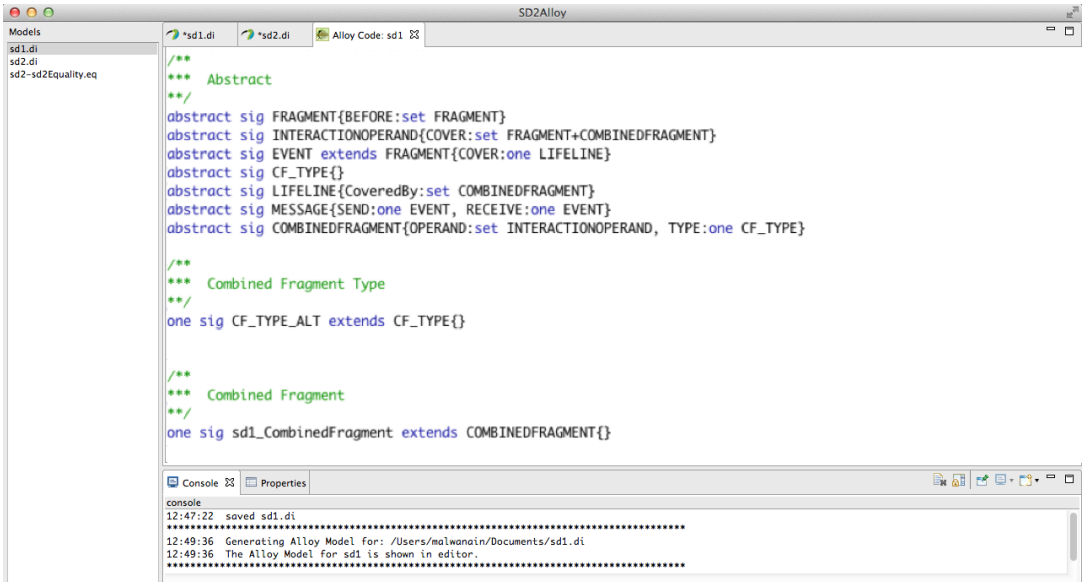


Figure A.5: Alloy code in SD2Alloy.

importing them. In this section, we will use the example in Figure 2.5, which has been drawn via *SD2Alloy*. Secondly, the sequence diagram is transformed into an Alloy model, thus allowing for powerful analysis to be conducted using the Alloy Analyser. The snippet of code below depicts part of the Alloy code generated using *SD2Alloy* from the sequence diagram in the running example. The complete Alloy code for the running example is presented in Appendix A.

```
abstract sig EVENT{NEXT:set EVENT, COVER:one LIFELINE}
abstract sig INTERACTIONOPERAND{COVER:set EVENT+COMBINEDFRAGMENT}
abstract sig CF_TYPE{}
abstract sig LIFELINE{CoveredBy:set COMBINEDFRAGMENT}
abstract sig MESSAGE{SEND:one MESSAGE_EVENT, RECEIVE:one MESSAGE_EVENT}
abstract sig COMBINEDFRAGMENT{OPERAND:set INTERACTIONOPERAND, TYPE:one CF_TYPE}


// Combined Fragment Type
one sig CF_TYPE_ALT extends CF_TYPE{}
one sig CF_TYPE_PAR extends CF_TYPE{}

//Combined Fragment
one sig SD1_CombinedFragment1 extends COMBINEDFRAGMENT{}
one sig SD1_CombinedFragment2 extends COMBINEDFRAGMENT{}
```

```
// Operand
one sig SD1_CombinedFragment1_Operand2 extends INTERACTIONOPERAND{}
one sig SD1_CombinedFragment1_Operand1 extends INTERACTIONOPERAND{}
lone sig SD1_CombinedFragment2_Operand2 extends INTERACTIONOPERAND{}
lone sig SD1_CombinedFragment2_Operand1 extends INTERACTIONOPERAND{}
```

A.9 Model Composition

Following the transformation of the sequence diagram, a compose process takes place. The developer use in this stage a Constraint Editor to add equalities for merging sequence diagrams. The editor allows the user to select the common elements of the current diagrams (figure A.6) and add expression, representing the equality between them as figure A.7 shows.

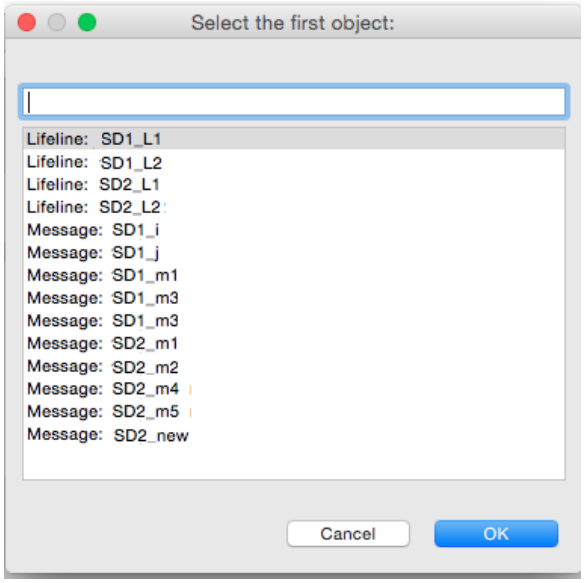


Figure A.6: List of models elements .

After the user has specified the equality elements, the tool merges them and produces a new Alloy model which corresponds to the union of all constraints associated to the input Alloy models and the glue contraints. Figure A.8 shows the merged Alloy model.

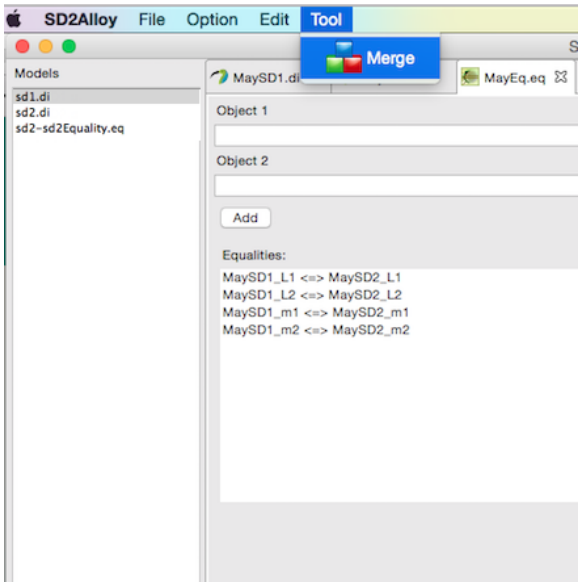


Figure A.7: Constraint Editor.

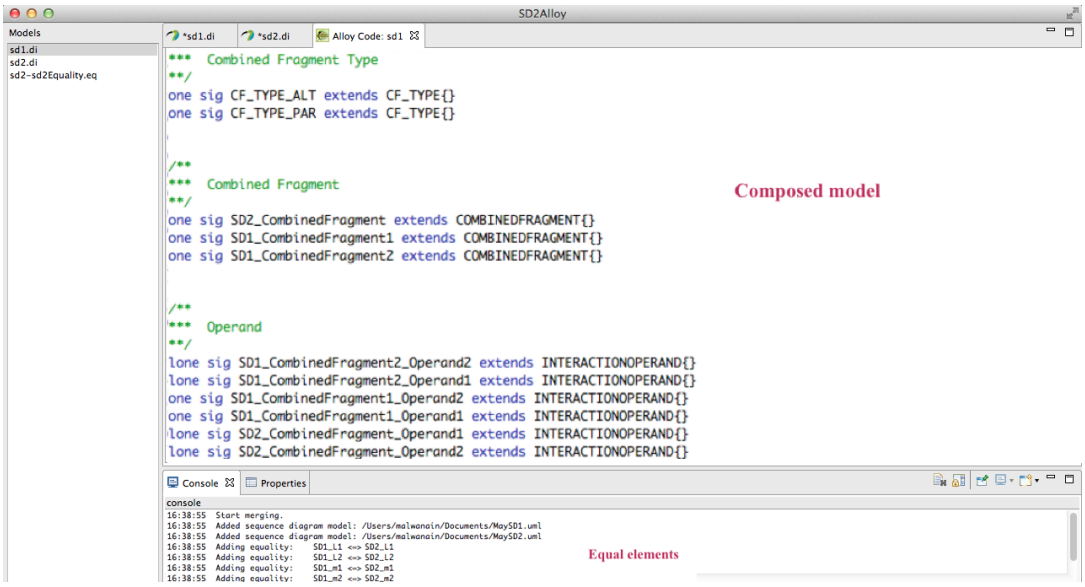


Figure A.8: composed model.

Solving this model via Alloy Analyser can produce a number of solutions (instances). These instances illustrate all possible solutions that may result from the composition of the sequence diagrams in Figures A.9.

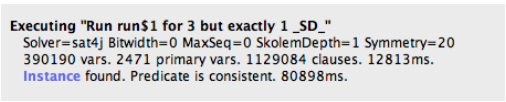


Figure A.9: composed summary in Alloy.

Figure A.9 presents a summary of the composition generated by the Alloy Analyser. It illustrates that the model satisfies all logical constraints, with the final model consisting of 390,190 variables and 1,129,084 clauses. The composition took 80.898 seconds to produce these results.

A.10 Chapter Summary

This chapter presents the implementation of the model composition tool, *SD2Alloy*. The description of the tool architecture is discussed in section 5.2. In the following sections, the technologies used in the development, such as the Papyrus tool and Sitra are described. Finally, section 5.7 provides a description of the *SD2Alloy* Eclipse plug-in.

APPENDIX B

ALLOY MODELS OF THE EXAMPLES IN CHAPTER 4

This section presents the complete Alloy textual code automatically generated from the *SD2ALLOY* relating to the running example presented in Chapter 4. Some comments have been manually written to explain which the code.

B.1 Alloy model for Sequence Diagram (sd1)

```
/**
***  Abstract signatures for the elements of the metamodel
**/

abstract sig INTERACTIONOPERAND{COVER:set EVENT+COMBINEDFRAGMENT}
abstract sig EVENT {COVER:one LIFELINE, Next:set EVENT}
abstract sig CF_TYPE{}
abstract sig LIFELINE{CoveredBy:set COMBINEDFRAGMENT}
abstract sig MESSAGE{SEND:one EVENT, RECEIVE:one EVENT}
abstract sig COMBINEDFRAGMENT{OPERAND:set INTERACTIONOPERAND, TYPE:one CF_TYPE}
/**
***  Combined Fragments Type
**/
one sig CF_TYPE_ALT extends CF_TYPE{}
one sig CF_TYPE_PAR extends CF_TYPE{}
/**
***  Combined Fragment
```

```
**/
one sig sd1_CF1 extends COMBINEDFRAGMENT{}
one sig sd1_CF2 extends COMBINEDFRAGMENT{}
/**
***  Operand
**////CombinedFragment Operands
one sig  sd1_CF1_Op2 extends INTERACTIONOPERAND{}
one sig  sd1_CF1_Op1 extends INTERACTIONOPERAND{}

lone sig sd1_CF2_Op2 extends INTERACTIONOPERAND{}
lone sig sd1_CF2_Op1 extends INTERACTIONOPERAND{}
/**
*** Messages Names
**/
one sig NAME_i, NAME_j, NAME_m1, NAME_m2, NAME_m3{}
/**
***  Lifelines Names
**/
one sig a, b{}

/**
***  Lifelines Classes
**/
one sig A, B{}
/**
***  Lifelines
**/
one sig sd1_a extends LIFELINE{NAME:one a, CLASS:one A}
one sig sd1_b extends LIFELINE{NAME:one b, CLASS:one B}

/**
***  Events
**/
one sig sd1_e2 extends EVENT{}
one sig sd1_e3 extends EVENT{}
lone sig sd1_e6 extends EVENT{}
lone sig sd1_e7 extends EVENT{}
one sig sd1_e9 extends EVENT{}
one sig sd1_g2 extends EVENT{}
one sig sd1_g3 extends EVENT{}
lone sig sd1_g6 extends EVENT{}
lone sig sd1_g7 extends EVENT{}
one sig sd1_g9 extends EVENT{}
```

```
/**
***  Messages
**/
lone sig sd1_m1 extends MESSAGE{NAME:one NAME_m1}
lone sig sd1_m2 extends MESSAGE{NAME:one NAME_m2}
one sig  sd1_m3 extends MESSAGE{NAME:one NAME_m3}
lone sig sd1_j extends MESSAGE{NAME:one NAME_j}
lone sig sd1_i extends MESSAGE{NAME:one NAME_i}
/**
***  Binding: Combined Fragment Type
**/
/// This fact define the type of each CombinedFragment

fact{
all _CF: sd1_CF2 | _CF.TYPE = CF_TYPE_ALT
all _CF: sd1_CF1 | _CF.TYPE = CF_TYPE_PAR
}
/**
***  Binding: Message->Event
**/
// This fact specify the event of message (send / receive)
fact{
sd1_i.SEND =  sd1_e2
sd1_i.RECEIVE =  sd1_g2
sd1_m1.SEND =  sd1_e3
sd1_m1.RECEIVE =  sd1_g3
sd1_j.SEND =  sd1_e6
sd1_j.RECEIVE =  sd1_g6
sd1_m2.SEND =  sd1_e7
sd1_m3.SEND =  sd1_e9
sd1_m2.RECEIVE =  sd1_g7
sd1_m3.RECEIVE =  sd1_g9
}
/**
***  Covering: Combined Fragment->Operand
**/
// This fact connect the Operands with their Combined Fragments
fact{
sd1_CF1_Op1 in sd1_CF1.OPERAND
sd1_CF1_Op2 in sd1_CF1.OPERAND

sd1_CF2_Op2 in sd1_CF2.OPERAND
sd1_CF2_Op1 in sd1_CF2.OPERAND
```

```
}
/**
***  Covering: Combined Fragment->Lifeline
**/
fact{
sd1_CF1 in sd1_a.CoveredBy
sd1_CF2 in sd1_a.CoveredBy
sd1_CF1 in sd1_b.CoveredBy
sd1_CF2 in sd1_b.CoveredBy
}
/**
***  Covering: Event->Lifeline
**/
// This fact connect the Events to Lifelines that covered by
fact{
all _E: sd1_e2 | _E.COVER=sd1_a
all _E: sd1_g2 | _E.COVER=sd1_b
all _E: sd1_e3 | _E.COVER=sd1_a
all _E: sd1_g3 | _E.COVER=sd1_b
all _E: sd1_e6 | _E.COVER=sd1_a
all _E: sd1_g6 | _E.COVER=sd1_b
all _E: sd1_e7 | _E.COVER=sd1_a
all _E: sd1_g7 | _E.COVER=sd1_b
all _E: sd1_e9 | _E.COVER=sd1_a
all _E: sd1_g9 | _E.COVER=sd1_b
}
/**
***  Covering: Operand->Fragment
**/
// This fact connect the Events or CombinedFragment to the operands that located inside or
    connect them directly to the Interaction if they are not nested any operand
fact{

all _F: sd1_e2 | _F in sd1_CF1_Op1.*(COVER.OPERAND).COVER
all _F: sd1_g2 | _F in sd1_CF1_Op1.*(COVER.OPERAND).COVER
all _F: sd1_e3 | _F in sd1_CF1_Op2.*(COVER.OPERAND).COVER
all _F: sd1_g3 | _F in sd1_CF1_Op2.*(COVER.OPERAND).COVER
all _F: sd1_e6 | _F in sd1_CF2_Op1.*(COVER.OPERAND).COVER
all _F: sd1_e6 | _F in sd1_CF2_Op1.*(COVER.OPERAND).COVER
all _F: sd1_e7 | _F in sd1_CF2_Op2.*(COVER.OPERAND).COVER
all _F: sd1_e7 | _F in sd1_CF2_Op2.*(COVER.OPERAND).COVER
all _F: sd1_e9 | _F in Interaction.*(COVER.OPERAND).COVER
all _F: sd1_e9 | _F !in sd1_CF2.^(OPERAND.COVER)
```

```
all _F: sd1_e9 | _F !in sd1_CF1.^(OPERAND.COVER)
all _F: sd1_g9 | _F !in sd1_CF1.^(OPERAND.COVER)
all _F: sd1_g9 | _F !in sd1_CF2.^(OPERAND.COVER)
all _F: sd1_g9 | _F in Interaction.*(COVER.OPERAND).COVER
all _F: sd1_CF1 | _F !in sd1_CF2.^(OPERAND.COVER)
all _F: sd1_CF2 | _F !in sd1_CF1.^(OPERAND.COVER)
all _F: sd1_CF1 | _F in Interaction.*(COVER.OPERAND).COVER
all _F: sd1_CF2 | _F in Interaction.*(COVER.OPERAND).COVER
}
/**
***  Number: Message = Operand
**/
//This fact for the  messages that located inside CombinedFragment. it means that each message
     must occur when its Operand appear.
fact{

#sd1_m1=#sd1_CF1_Op2
#sd1_j=#sd1_CF2_Op1
#sd1_i=#sd1_CF1_Op1
#sd1_m2=#sd1_CF2_Op2
}
/**
***  GeneralOrdering
**/
fact GeneralOrder {
// The content of CombinedFragment1(i,M1 messages) before The content of CombinedFragment2(j,
    M2 messages)
all l: sd1_a + sd1_b , ev1:sd1_CF1.OPERAND.COVER, ev2:sd1_CF2.OPERAND.COVER | ev1.COVER = l
    and ev2.COVER = l => ev2 in ev1.^ Next
//CombinedFragment2 before Message M3
all l: sd1_a , ev1:sd1_CF2.OPERAND.COVER , ev2:sd1_e9 | ev1.COVER = l => ev2 in ev1.^ Next
all l: sd1_b , ev1:sd1_CF2.OPERAND.COVER , ev2:sd1_g9 | ev1.COVER = l => ev2 in ev1.^ Next}
/**
***  SD interaction
**/
one sig Interaction extends INTERACTIONOPERAND{
LIFELINES: Interaction one -> LIFELINE,
COMBINEDFRAGMENTS: Interaction one -> COMBINEDFRAGMENT,
MESSAGES: Interaction one -> MESSAGE}
/**
***  Constraint: Lifeline
**/
// one event can have at most one Next one one lifeline
```

```
fact{all _E1: EVENT | lone _E2: EVENT-_E1 | _E2 in _E1.Next and _E2.COVER=_E1.COVER}
fact{all _E1: EVENT | lone _E2: EVENT-_E1 | _E1 in _E2.Next and _E2.COVER=_E1.COVER}
// at most one event can have no next on the same lifeline
fact{all _L: LIFELINE | lone _E1: EVENT | _E1.COVER=_L and (_L !in _E1.Next.COVER or #_E1.Next
    =0) }
/**
***  Constraint: Combined Fragment
**/
// one CF should be covered by at most one Operand
fact{all _F: EVENT  | lone _OP: INTERACTIONOPERAND | _F in _OP.COVER}
fact{all _OP: INTERACTIONOPERAND | lone _F: COMBINEDFRAGMENT | _OP in _F.OPERAND}
// INTERACTIONOPERAND: the children can not cover their parent
fact{all _OP: INTERACTIONOPERAND | _OP !in _OP.^(COVER.OPERAND)}
// one cf can be cover by at most one op
fact{all _CF: COMBINEDFRAGMENT | one _OP: INTERACTIONOPERAND | _CF in _OP.COVER}
// INTERACTIONOPERAND: in one OP, at most one event for each lifeline can have no Next
fact{all _L: LIFELINE, _OP: INTERACTIONOPERAND | lone _E: EVENT | _E in _OP.COVER and _E.COVER
    =_L and #_E.Next=0}
// alt: exact one operand will be executed
fact{all _CF: COMBINEDFRAGMENT | (_CF.TYPE = CF_TYPE_ALT) => #_CF.OPERAND = 1}
// INTERACTIONOPERAND: one OP can not be before and after the same other OP
fact{all _OP1: INTERACTIONOPERAND, _OP2:INTERACTIONOPERAND, _E1: _OP1.COVER| ( _E1 in _OP2.*(
    COVER.OPERAND).COVER.*Next and  _OP1 != _OP2 and _OP1 != Interaction and _OP2 !=
    Interaction) => _OP1.COVER in _OP2.*(COVER.OPERAND).COVER.*Next}
/**
***  Constraint: Message
**/
// one event can be send/receive by at most one message
fact{all _E: EVENT | one _M: MESSAGE | _E = _M.SEND or _E = _M.RECEIVE}
// only allow relation between Events either they are in same message or on same lifeline
fact{all _E1: EVENT, _M: MESSAGE,  _E2: EVENT | (_E1 in _M.SEND and _E2 in _E1.Next) => (_M.
    RECEIVE=_E2) or (_E1.COVER=_E2.COVER)}
fact{all _E1: EVENT, _M: MESSAGE,  _E2: EVENT | (_E1 in _M.RECEIVE and _E2 in _E1.Next) => (
    _E1.COVER=_E2.COVER)}
// one message's send/receive should be covered by the same operand
fact{all _M: MESSAGE | one _OP: INTERACTIONOPERAND | _M.SEND in _OP.COVER and _M.RECEIVE in
    _OP.COVER}
// send before receive
fact{all _M: MESSAGE | _M.RECEIVE in _M.SEND.Next}
// no circle
fact{no e:EVENT  |  e in e.^Next}
/**
***  Run
```

```
**/
run{}
```

B.2 Alloy model for Sequence Diagram (sd2)

```
/**
***  Abstract signatures for the elements of the models(metamodel)
**/
abstract sig INTERACTIONOPERAND{COVER:set EVENT+COMBINEDFRAGMENT}
abstract sig EVENT {COVER:one LIFELINE, Next:set EVENT}
abstract sig CF_TYPE{}
abstract sig LIFELINE{CoveredBy:set COMBINEDFRAGMENT}
abstract sig MESSAGE{SEND:one EVENT, RECEIVE:one EVENT}
abstract sig COMBINEDFRAGMENT{OPERAND:set INTERACTIONOPERAND, TYPE:one CF_TYPE}
/**
***  Combined Fragment Type which is Alt in this example
**/
one sig CF_TYPE_ALT extends CF_TYPE{}
/**
***  Combined Fragment
**/
one sig sd2_CF extends COMBINEDFRAGMENT{}
/**
***  Operand
**///CombinedFragment Operands
lone sig sd2_Op1 extends INTERACTIONOPERAND{}
lone sig sd2_Op2 extends INTERACTIONOPERAND{}
/**
*** Messages Names
**/
one sig  Name_m1, Name_m2, Name_new, Name_m4, Name_m5{}


/**
***  Lifelines Names
**/
one sig a, b{}
/**
***  Lifelines Classes
**/
one sig A, B{}
```

```
/**
***  Lifelines
**/
one sig sd2_a extends LIFELINE{NAME:one a, CLASS:one A}
one sig sd2_b extends LIFELINE{NAME:one b, CLASS:one B}
/**
***  Events
**/
one sig sd2_f1 extends EVENT{}
one sig sd2_h1 extends EVENT{}
one sig sd2_h2 extends EVENT{}
one sig sd2_f2 extends EVENT{}
one sig sd2_f3 extends EVENT{}
one sig sd2_h3 extends EVENT{}
lone sig sd2_f6 extends EVENT{}
lone sig sd2_h6 extends EVENT{}
lone sig sd2_f5 extends EVENT{}
lone sig sd2_h5 extends EVENT{}
/**
***  Messages
**/
one sig sd2_new extends MESSAGE{NAME:one Name_new}
one sig sd2_m1 extends MESSAGE{NAME:one Name_m1}
one sig sd2_m2 extends MESSAGE{NAME:one Name_m2}
lone sig sd2_m4 extends MESSAGE{NAME:one  Name_m4}
lone sig sd2_m5 extends MESSAGE{NAME:one  Name_m5}
/**
***  Binding: Combined Fragment Type
**/
// This fact define the type of each CombinedFragment
fact{
all _CF: sd2_CF | _CF.TYPE = CF_TYPE_ALT
}
/**
***  Binding: Message->Event
**/
// This fact specify the event of message send and receive
fact{
sd2_m4.SEND = sd2_f5
sd2_m4.RECEIVE = sd2_h
sd2_new.SEND = sd2_h2
sd2_new.RECEIVE = sd2_f2
sd2_m5.SEND = sd2_f6
```

```
sd2_m5.RECEIVE = sd2_h6
sd2_m2.SEND = sd2_f3
sd2_m2.RECEIVE = sd2_h3
sd2_m1.SEND = sd2_f1
sd2_m1.RECEIVE = sd2_h1
}
/**
***  Covering: Combined Fragment->Operand
**/
// This fact connect the Operands with their Combined Fragments
fact{
sd2_Op2 in sd2_CF.OPERAND
sd2_Op1 in sd2_CF.OPERAND
}
/**
***  Covering: Combined Fragment->Lifeline
**/
fact{
sd2_CF in sd2_a.CoveredBy
sd2_CF in sd2_b.CoveredBy
}
/**
***  Covering: Event->Lifeline
**/
// This fact connect the Events to Lifelines that covered by
fact{
all _E: sd2_f5 | _E.COVER= sd2_a
all _E: sd2_h6 | _E.COVER= sd2_b
all _E: sd2_f1 | _E.COVER= sd2_a
all _E: sd2_f2 | _E.COVER= sd2_a
all _E: sd2_f6 | _E.COVER= sd2_a
all _E: sd2_h3 | _E.COVER= sd2_b
all _E: sd2_h5 | _E.COVER= sd2_b
all _E: sd2_f3 | _E.COVER= sd2_a
all _E: sd2_h2 | _E.COVER= sd2_b
all _E: sd2_h1 | _E.COVER= sd2_b
}
/**
***  Covering: Operand->Fragment
**/
// This fact connect the Events or CombinedFragment to the operands that located inside or
    connect them directly to the Interaction if they are not nested any operand
fact{
```

```
all _F:  sd2_f1 | _F in Interaction.*(COVER.OPERAND).COVER
all _F:  sd2_f1 | _F !in  sd2_CF.^(OPERAND.COVER)
all _F:  sd2_h1 | _F !in  sd2_CF.^(OPERAND.COVER)
all _F:  sd2_h1 | _F in Interaction.*(COVER.OPERAND).COVER
all _F:  sd2_h2 | _F !in  sd2_CF.^(OPERAND.COVER)
all _F:  sd2_h2 | _F in Interaction.*(COVER.OPERAND).COVER
all _F:  sd2_f2 | _F !in  sd2_CF.^(OPERAND.COVER)
all _F:  sd2_f2 | _F in Interaction.*(COVER.OPERAND).COVER
all _F:  sd2_f3 | _F !in  sd2_CF.^(OPERAND.COVER)
all _F:  sd2_f3 | _F in Interaction.*(COVER.OPERAND).COVER
all _F:  sd2_h3 | _F in Interaction.*(COVER.OPERAND).COVER
all _F:  sd2_h3 | _F !in  sd2_CF.^(OPERAND.COVER)
all _F:  sd2_f6 | _F in  sd2_Op1.*(COVER.OPERAND).COVER
all _F:  sd2_h6 | _F in   sd2_Op1.*(COVER.OPERAND).COVER
all _F:  sd2_f5 | _F in  sd2_Op2.*(COVER.OPERAND).COVER
all _F:  sd2_h5 | _F in  sd2_Op2.*(COVER.OPERAND).COVER
all _F:  sd2_CF | _F in Interaction.*(COVER.OPERAND).COVER
}
/**
***  Number: Message = Operand
**/
//This fact for the  messages that located inside CombinedFragment. it means that each message
     must occur when its Operand appear.
fact{
#sd2_m4=# sd2_Op2
#sd2_m5=# sd2_Op1
}

/**
***  GeneralOrdering
**/

fact{
//Message M1 before new
all ev1: sd2_f2,  ev2: sd2_f1 |  ev1 in  ev2.^Next
all ev1: sd2_h2,  ev2: sd2_h1 | ev1 in ev2.^Next
//Message new before M2
all ev1: sd2_f3,  ev2: sd2_f2 | ev1 in ev2.^Next
all ev1: sd2_h3,  ev2: sd2_h2 | ev1 in ev2.^Next
//Message M2 before the content of SD2_CombinedFragment (yes or no messages)
all l: sd2_a, ev1: sd2_f3, ev2: sd2_CF.OPERAND.COVER | ev2.COVER=l =>ev2 in ev1.^Next
all l: sd2_b, ev1: sd2_h3, ev2: sd2_CF.OPERAND.COVER | ev2.COVER=l => ev2 in ev1.^Next
}
```

```
/**
***  SD interaction
**/
one sig Interaction extends INTERACTIONOPERAND{
LIFELINES: Interaction one -> LIFELINE,
COMBINEDFRAGMENTS: Interaction one -> COMBINEDFRAGMENT,
MESSAGES: Interaction one -> MESSAGE}
/**
***  Constraint: Lifeline
**/
// one event can have at most one Next one one lifeline
fact{all _E1: EVENT | lone _E2: EVENT-_E1 | _E2 in _E1.Next and _E2.COVER=_E1.COVER}
fact{all _E1: EVENT | lone _E2: EVENT-_E1 | _E1 in _E2.Next and _E2.COVER=_E1.COVER}
// at most one event can have no next on the same lifeline
fact{all _L: LIFELINE | lone _E1: EVENT | _E1.COVER=_L and (_L !in _E1.Next.COVER or #_E1.Next
    =0) }
/**
***  Constraint: Combined Fragment
**/
// one CF should be covered by at most one Operand
fact{all _F: EVENT  | lone _OP: INTERACTIONOPERAND | _F in _OP.COVER}
fact{all _OP: INTERACTIONOPERAND | lone _F: COMBINEDFRAGMENT | _OP in _F.OPERAND}
// INTERACTIONOPERAND: the children can not cover their parent
fact{all _OP: INTERACTIONOPERAND | _OP !in _OP.^(COVER.OPERAND)}
// one cf can be cover by at most one op
fact{all _CF: COMBINEDFRAGMENT | one _OP: INTERACTIONOPERAND | _CF in _OP.COVER}
// INTERACTIONOPERAND: in one OP, at most one event for each lifeline can have no Next
fact{all _L: LIFELINE, _OP: INTERACTIONOPERAND | lone _E: EVENT | _E in _OP.COVER and _E.COVER
    =_L and #_E.Next=0}
// alt: exact one operand will be executed
fact{all _CF: COMBINEDFRAGMENT | (_CF.TYPE = CF_TYPE_ALT) => #_CF.OPERAND = 1}
// INTERACTIONOPERAND: one OP can not be before and after the same other OP
fact{all _OP1: INTERACTIONOPERAND, _OP2:INTERACTIONOPERAND, _E1: _OP1.COVER| ( _E1 in _OP2.*(
    COVER.OPERAND).COVER.*Next and  _OP1 != _OP2 and _OP1 != Interaction and _OP2 !=
    Interaction) => _OP1.COVER in _OP2.*(COVER.OPERAND).COVER.*Next}
/**
***  Constraint: Message
**/
// one event can be send/receive by at most one message
fact{all _E: EVENT | one _M: MESSAGE | _E = _M.SEND or _E = _M.RECEIVE}
// only allow relation between Events either they are in same message or on same lifeline
```

```
fact{all _E1: EVENT, _M: MESSAGE,  _E2: EVENT | (_E1 in _M.SEND and _E2 in _E1.Next) => (_M.
    RECEIVE=_E2) or (_E1.COVER=_E2.COVER)}
fact{all _E1: EVENT, _M: MESSAGE,  _E2: EVENT | (_E1 in _M.RECEIVE and _E2 in _E1.Next) => (
    _E1.COVER=_E2.COVER)}
// one message's send/receive should be covered by the same operand
fact{all _M: MESSAGE | one _OP: INTERACTIONOPERAND | _M.SEND in _OP.COVER and _M.RECEIVE in
    _OP.COVER}
// send before receive
fact{all _M: MESSAGE | _M.RECEIVE in _M.SEND.Next}
/**
***  Constraint: Fragment
**/
// no circle
fact {no e:EVENT  |  e in e.^Next}
/**
***  Run
**/
run{}
```

B.3 Alloy model for the composition of sd1 and sd2 (sd3)

```
/**
***  Abstract signatures for the elements of the metamodel
**/
abstract sig INTERACTIONOPERAND{COVER:set EVENT+COMBINEDFRAGMENT}
abstract sig EVENT {COVER:one LIFELINE, Next:set EVENT}
abstract sig CF_TYPE{}
abstract sig LIFELINE{CoveredBy:set COMBINEDFRAGMENT}
abstract sig MESSAGE{SEND:one EVENT, RECEIVE:one EVENT}
abstract sig COMBINEDFRAGMENT{OPERAND:set INTERACTIONOPERAND, TYPE:one CF_TYPE}
/**
***  Combined Fragments Types
**/
one sig CF_TYPE_ALT extends CF_TYPE{}
one sig CF_TYPE_PAR extends CF_TYPE{}
/**
***  Combined Fragment
**/
//This CombinedFragment for the second diagram which contains yes or no
lone sig sd2_CF extends COMBINEDFRAGMENT{}
```

```
//This CombinedFragment for the second diagram which contains M1 or a
one sig sd1_CF1 extends COMBINEDFRAGMENT{}
//This CombinedFragment for the second diagram which contains M2 or b
one sig sd1_CF2 extends COMBINEDFRAGMENT{}
/**
***  Operands
**/
//CombinedFragment2 Operands of SD1 diagram
lone sig sd1_CF2_Op2 extends INTERACTIONOPERAND{}
lone sig sd1_CF2_Op1 extends INTERACTIONOPERAND{}
//CombinedFragment1 Operands of SD1 diagram
one sig  sd1_CF1_Op2 extends INTERACTIONOPERAND{}
one sig sd1_CF1_Op1 extends INTERACTIONOPERAND{}
//CombinedFragment Operands of SD2 diagram
lone sig sd2_CF_Op1 extends INTERACTIONOPERAND{}
lone sig sd2_CF_Op2 extends INTERACTIONOPERAND{}
/**
***  Messages Names
**/
one sig m1, m2, m3, m4, m5, new, i, j{}
/**
***  Lifelines Names
**/
one sig a, b{}
/**
***  Lifelines Classes
**/
one sig A, B{}
/**
***  Lifelines
**/
//SD2 lifelines
lone sig sd2_b extends LIFELINE{NAME:one b, CLASS:one B}
lone sig sd2_a extends LIFELINE{NAME:one a, CLASS:one A}
//SD1 lifelines
one sig sd1_a extends LIFELINE{NAME:one a, CLASS:one A}
one sig sd1_b extends LIFELINE{NAME:one b, CLASS:one B}
/**
***  Event
**/
//SD1 Events
lone sig sd1_e2 extends EVENT{}
lone sig sd1_e3 extends EVENT{}
```

```
lone sig sd1_e6 extends EVENT{}
lone sig sd1_e7 extends EVENT{}
lone sig sd1_e9 extends EVENT{}
lone sig sd1_g2 extends EVENT{}
lone sig sd1_g3 extends EVENT{}
lone sig sd1_g6 extends EVENT{}
lone sig sd1_g7 extends EVENT{}
lone sig sd1_g9 extends EVENT{}

//SD2 Events
lone sig sd2_f1 extends EVENT{}
lone sig sd2_h1 extends EVENT{}
one sig sd2_h2 extends EVENT{}
one sig sd2_f2 extends EVENT{}
lone sig sd2_f3 extends EVENT{}
lone sig sd2_h3 extends EVENT{}
lone sig sd2_f6 extends EVENT{}
lone sig sd2_h6 extends EVENT{}
lone sig sd2_f5 extends EVENT{}
lone sig sd2_h5 extends EVENT{}
/**
***  Message
**/
//Messages of SD1 diagram
lone sig sd1_m1 extends MESSAGE{NAME:one m1}
lone sig sd1_m2 extends MESSAGE{NAME:one m2}
lone sig sd1_j extends MESSAGE{NAME:one j}
lone sig sd1_i extends MESSAGE{NAME:one i}
one sig sd1_m3 extends MESSAGE{NAME:one m3}

//Messages of  SD2 diagram
lone sig sd2_m1 extends MESSAGE{NAME:one m1}
one sig sd2_new extends MESSAGE{NAME:one new}
lone sig sd2_m2 extends MESSAGE{NAME:one m2}
lone sig sd2_m4 extends MESSAGE{NAME:one m4}
lone sig sd2_m5 extends MESSAGE{NAME:one m5}
/**
***  Binding: Combined Fragment Type
**/
// This fact define the type of each CombinedFragment
fact{
all _CF:sd1_CF2 | _CF.TYPE = CF_TYPE_ALT
all _CF:sd1_CF1 | _CF.TYPE = CF_TYPE_PAR
```

```
all _CF:sd2_CF | _CF.TYPE = CF_TYPE_ALT
}
/**
***  Binding: Message->Event
**/
// This fact specify the event of message send and receive
fact{
sd1_i.SEND = sd1_e2
sd1_i.RECEIVE = sd1_g2
sd1_m1.SEND = sd1_e3
sd1_m1.RECEIVE = sd1_g3
sd1_j.SEND = sd1_e6
sd1_j.RECEIVE = sd1_g6
sd1_m2.SEND = sd1_e7
sd1_m2.RECEIVE = sd1_g7
sd1_m3.SEND = sd1_e9
sd1_m3.RECEIVE = sd1_g9

sd2_m4.SEND = sd2_f6
sd2_m4.RECEIVE = sd2_h6
sd2_m5.SEND = sd2_f5
sd2_m5.RECEIVE = sd2_h5
sd2_new.SEND = sd2_h2
sd2_new.RECEIVE = sd2_f2
sd2_m2.SEND = sd2_f3
sd2_m2.RECEIVE = sd2_h3
sd2_m1.SEND = sd2_f1
sd2_m1.RECEIVE = sd2_h1
}
/**
***  Covering: Combined Fragment->Operand
**/
// This fact connect the Operands with their Combined Fragments
fact{

sd1_CF1_Op1 in sd1_CF1.OPERAND
sd1_CF1_Op2 in sd1_CF1.OPERAND
sd2_CF_Op1 in sd2_CF.OPERAND
sd2_CF_Op2 in sd2_CF.OPERAND
sd1_CF2_Op1 in sd1_CF2.OPERAND
sd1_CF2_Op2 in sd1_CF2.OPERAND
}
/**
```

```
***  Covering: Combined Fragment->Lifeline
**/
fact{
sd1_CF2 in sd1_a.CoveredBy
sd1_CF2 in sd1_b.CoveredBy
sd1_CF1 in sd1_a.CoveredBy
sd1_CF1 in sd1_b.CoveredBy
sd2_CF in sd1_b.CoveredBy
sd2_CF in sd1_a.CoveredBy
}
/**
***  Covering: Event->Lifeline
**/
// This fact connect the Events to Lifelines that covered by
fact{
all _E:sd1_e2 | _E.COVER=sd1_a
all _E:sd1_g2 | _E.COVER=sd1_b
all _E:sd1_e3 | _E.COVER=sd1_a
all _E:sd1_g3 | _E.COVER=sd1_b
all _E:sd1_e6 | _E.COVER=sd1_a
all _E:sd1_g6 | _E.COVER=sd1_b
all _E:sd1_e7 | _E.COVER=sd1_a
all _E:sd1_g7 | _E.COVER=sd1_b
all _E:sd1_e9 | _E.COVER=sd1_a
all _E:sd1_g9 | _E.COVER=sd1_b
all _E:sd2_h2 | _E.COVER=sd1_b
all _E:sd2_f2 | _E.COVER=sd1_a
all _E:sd2_f6 | _E.COVER=sd1_a
all _E:sd2_h6 | _E.COVER=sd1_b
all _E:sd2_f5 | _E.COVER=sd1_a
all _E:sd2_h5 | _E.COVER=sd1_b
all _E: sd2_h3 | _E.COVER= sd2_b
all _E: sd2_f3 | _E.COVER= sd2_a
all _E: sd2_h1 | _E.COVER= sd2_b
all _E: sd2_f1 | _E.COVER= sd2_a
}
/**
***  Covering: Operand->Fragment
**/
// This fact connect the Events or CombinedFragment to the operands that located inside or
    connect them directly to the Interaction if they are not nested any operand
fact{
all _F: sd2_CF | _F in Interaction.*(COVER.OPERAND).COVER
```

```
all _F: sd2_h2 | _F in Interaction.*(COVER.OPERAND).COVER
all _F: sd1_g9 | _F !in sd1_CF2.^(OPERAND.COVER)
all _F: sd2_f2 | _F !in sd2_CF.^(OPERAND.COVER)
all _F: sd1_e9 | _F in Interaction.*(COVER.OPERAND).COVER
all _F: sd1_e9 | _F !in sd1_CF2.^(OPERAND.COVER)
all _F: sd1_CF1 | _F !in sd1_CF2.^(OPERAND.COVER)
all _F: sd1_e7 | _F in Interaction.*(COVER.OPERAND).COVER
all _F: sd1_g6 | _F in sd1_CF2_Op1.*(COVER.OPERAND).COVER
all _F: sd1_g3 | _F in sd1_CF1_Op2.*(COVER.OPERAND).COVER
all _F: sd2_h5 | _F in sd2_CF_Op2.*(COVER.OPERAND).COVER
all _F: sd1_e6 | _F in sd1_CF2_Op1.*(COVER.OPERAND).COVER
all _F: sd2_h2 | _F !in sd2_CF.^(OPERAND.COVER)
all _F: sd1_e3 | _F in Interaction.*(COVER.OPERAND).COVER
all _F: sd1_g7 | _F !in sd2_CF.^(OPERAND.COVER)
all _F: sd1_g9 | _F !in sd1_CF1.^(OPERAND.COVER)
all _F: sd1_g3 | _F !in sd2_CF.^(OPERAND.COVER)
all _F: sd1_e7 | _F in sd1_CF2_Op2.*(COVER.OPERAND).COVER
all _F: sd1_CF2 | _F !in sd1_CF1.^(OPERAND.COVER)
all _F: sd1_CF1 | _F in Interaction.*(COVER.OPERAND).COVER
all _F: sd2_f2 | _F in Interaction.*(COVER.OPERAND).COVER
all _F: sd1_e2| _F in sd1_CF1_Op1.*(COVER.OPERAND).COVER
all _F: sd1_e7 | _F !in sd2_CF.^(OPERAND.COVER)
all _F: sd1_g7 | _F in sd1_CF2_Op2.*(COVER.OPERAND).COVER
all _F: sd1_CF2 | _F in Interaction.*(COVER.OPERAND).COVER
all _F: sd1_g7 | _F in Interaction.*(COVER.OPERAND).COVER
all _F: sd2_h6 | _F in sd2_CF_Op1.*(COVER.OPERAND).COVER
all _F: sd2_f6 | _F in sd2_CF_Op1.*(COVER.OPERAND).COVER
all _F: sd1_g3 | _F in Interaction.*(COVER.OPERAND).COVER
all _F: sd1_e3 | _F in sd1_CF1_Op2.*(COVER.OPERAND).COVER
all _F: sd1_e9 | _F !in sd1_CF1.^(OPERAND.COVER)
all _F: sd2_f5 | _F in sd2_CF_Op2.*(COVER.OPERAND).COVER
all _F: sd1_g2 | _F in sd1_CF1_Op1.*(COVER.OPERAND).COVER
all _F: sd1_e3 | _F !in sd2_CF.^(OPERAND.COVER)
all _F: sd1_g9 | _F in Interaction.*(COVER.OPERAND).COVER
all _F:  sd2_f1 | _F in Interaction.*(COVER.OPERAND).COVER
all _F:  sd2_f1 | _F !in  sd2_CF.^(OPERAND.COVER)
all _F:  sd2_h1 | _F !in  sd2_CF.^(OPERAND.COVER)
all _F:  sd2_h1 | _F in Interaction.*(COVER.OPERAND).COVER
all _F:  sd2_f3 | _F !in  sd2_CF.^(OPERAND.COVER)
all _F:  sd2_f3 | _F in Interaction.*(COVER.OPERAND).COVER
all _F:  sd2_h3 | _F in Interaction.*(COVER.OPERAND).COVER
all _F:  sd2_h3 | _F !in  sd2_CF.^(OPERAND.COVER)
}
```

```
/**
***  Number: Message = Operand
**/
//This fact for the  messages that located inside (Alt) CombinedFragment. it means that each
    message must occur when its Operand appear or the message
//occure when the message before occur.
fact{
#sd2_m4=#sd2_CF_Op1
#sd2_m5=#sd2_CF_Op2
#sd1_m1=#sd1_CF1_Op2
#sd1_m2=#sd1_CF2_Op2
#sd1_j=#sd1_CF2_Op1
#sd1_i=#sd1_CF1_Op1
#sd2_new=#sd1_m1
#sd2_CF = #sd1_m2
}
/**
***  GeneralOrdering
**/
fact{
//new before M2
all ev1:sd1_g7, ev2:sd2_h2 | ev1 in ev2.^Next
all ev1:sd1_e7, ev2:sd2_f2 | ev1 in ev2.^Next

//M1 before new
all ev1:sd2_h2, ev2:sd1_g3 | ev1 in ev2.^Next
all ev1:sd2_f2, ev2:sd1_e3 | ev1 in ev2.^Next

//SD1_CombinedFragment before SD1_CombinedFragment2
all ev1:sd1_CF1.OPERAND.COVER, ev2:sd1_CF2.OPERAND.COVER |  ev2 in ev1.^Next

//// M2 before SD2_CombinedFragment
all l:sd1_a, ev1:sd1_e7, ev2:sd2_CF.OPERAND.COVER | ev2.COVER=l => ev2 in ev1.^Next
all l:sd1_b, ev1:sd1_g7, ev2:sd2_CF.OPERAND.COVER | ev2.COVER=l => ev2 in ev1.^Next

//M3 after  SD1_CombinedFragment2
all l:sd1_a, ev1:sd1_CF2.OPERAND.COVER, ev2:sd1_e9 | ev1.COVER=l => ev2 in ev1.^Next
all l:sd1_b, ev1:sd1_CF2.OPERAND.COVER, ev2:sd1_g9 | ev1.COVER=l => ev2 in ev1.^Next

}
/**
***  Glue
**/
```

```
fact{
//Merging message M1 of SD2 diagram with its events
all SD1M:sd1_m1 , SD2M:sd2_m1 | (SD1M.NAME = SD2M.NAME) => # SD2M = 0
# sd2_h1 = 0
# sd2_f1 = 0
//Merging message M2 of SD2 diagram with its events
all SD1M:sd1_m2 , SD2M:sd2_m2 | (SD1M.NAME = SD2M.NAME) => # SD2M = 0
# sd2_h3 = 0
# sd2_f3 = 0

//Merging  Lifeline L1 of SD2 diagram
all SD1L:sd1_a , SD2L:sd2_a | (SD1L.NAME = SD2L.NAME && SD1L.CLASS = SD2L.CLASS) => # SD2L = 0
//Merging  Lifeline L2 of SD2 diagram
all SD1L:sd1_b , SD2L:sd2_b | (SD1L.NAME = SD2L.NAME && SD1L.CLASS = SD2L.CLASS) => # SD2L = 0

}
/**
***  Behavioural  Glue
**/

fact negativeTrace {#sd1_j=0
all sd1_j_send:sd1_e6, sd1_j_receive:sd1_g6 |
#sd1_j_send=0 and #sd1_j_receive=0}

fact occurrence {
#sd1_m3.SEND =#sd1_m2.SEND and #sd1_m3.RECEIVE =# sd1_m2.RECEIVE

all sd1_m2_send:sd1_e7, sd1_m3_send:sd1_e9 | sd1_m3_send in sd1_m2_send.^Next
all sd1_m2_receive:sd1_g7, sd1_m3_receive:sd1_g9 | sd1_m3_receive in sd1_m2_receive.^Next}
/**
***  SD interaction
**/
one sig Interaction extends INTERACTIONOPERAND{
LIFELINES: Interaction one -> LIFELINE,
COMBINEDFRAGMENTS: Interaction one -> COMBINEDFRAGMENT,
MESSAGES: Interaction one -> MESSAGE}
/**
***  Constraint: Lifeline
**/
// one event can have at most one Next one one lifeline
fact{all _E1: EVENT | lone _E2: EVENT-_E1 | _E2 in _E1.Next and _E2.COVER=_E1.COVER}
fact{all _E1: EVENT | lone _E2: EVENT-_E1 | _E1 in _E2.Next and _E2.COVER=_E1.COVER}
// at most one event can have no next on the same lifeline
```

```
fact{all _L: LIFELINE | lone _E1: EVENT | _E1.COVER=_L and (_L !in _E1.Next.COVER or #_E1.Next
    =0) }
/**
***  Constraint: Combined Fragment
**/
// one CF should be covered by at most one Operand
fact{all _F: EVENT  | lone _OP: INTERACTIONOPERAND | _F in _OP.COVER}
fact{all _OP: INTERACTIONOPERAND | lone _F: COMBINEDFRAGMENT | _OP in _F.OPERAND}
// INTERACTIONOPERAND: the children can not cover their parent
fact{all _OP: INTERACTIONOPERAND | _OP !in _OP.^(COVER.OPERAND)}
// one cf can be cover by at most one op
fact{all _CF: COMBINEDFRAGMENT | one _OP: INTERACTIONOPERAND | _CF in _OP.COVER}
// INTERACTIONOPERAND: in one OP, at most one event for each lifeline can have no Next
fact{all _L: LIFELINE, _OP: INTERACTIONOPERAND | lone _E: EVENT | _E in _OP.COVER and _E.COVER
    =_L and #_E.Next=0}
// alt: exact one operand will be executed
fact{all _CF: COMBINEDFRAGMENT | (_CF.TYPE = CF_TYPE_ALT) => #_CF.OPERAND = 1}
// INTERACTIONOPERAND: one OP can not be before and after the same other OP
fact{all _OP1: INTERACTIONOPERAND, _OP2:INTERACTIONOPERAND, _E1: _OP1.COVER| ( _E1 in _OP2.*(
    COVER.OPERAND).COVER.*Next and  _OP1 != _OP2 and _OP1 != Interaction and _OP2 !=
    Interaction) => _OP1.COVER in _OP2.*(COVER.OPERAND).COVER.*Next}
/**
***  Constraint: Message
**/
// one event can be send/receive by at most one message
fact{all _E: EVENT | one _M: MESSAGE | _E = _M.SEND or _E = _M.RECEIVE}
// only allow relation between Events either they are in same message or on same lifeline
fact{all _E1: EVENT, _M: MESSAGE,  _E2: EVENT | (_E1 in _M.SEND and _E2 in _E1.Next) => (_M.
    RECEIVE=_E2) or (_E1.COVER=_E2.COVER)}
fact{all _E1: EVENT, _M: MESSAGE,  _E2: EVENT | (_E1 in _M.RECEIVE and _E2 in _E1.Next) => (
    _E1.COVER=_E2.COVER)}
// one message's send/receive should be covered by the same operand
fact{all _M: MESSAGE | one _OP: INTERACTIONOPERAND | _M.SEND in _OP.COVER and _M.RECEIVE in
    _OP.COVER}
// send before receive
fact{all _M: MESSAGE | _M.RECEIVE in _M.SEND.Next}
// no circle
fact {no e:EVENT  |  e in e.^Next}
/**
***  Run
**/
run{}
```

APPENDIX C

Z3 CODE OF THE EXAMPLES IN CHAPTER 6

This section presents the complete Z3 textual code for the running example presented in Chapter 6. This code represents the sequence diagrams sd1 and sd2 and the composition (see Figure2.5). In addition, this section illustrates the Z3 code for petrol station example explained in Chapter 6 section 6.4.

C.1 Z3 code for Sequence Diagram (sd1)

```
from z3 import *
#---------------------------------
#lifelines declarations
#---------------------------------
l = DeclareSort('l')
Sd1_a = Const('Sd1_a', l)
Sd1_b = Const('Sd1_b', l)
L_i = Const('L_i', l)
L_j = Const('L_j', l)

#---------------------------------
#Lifeline_name declarations
#---------------------------------
Lifeline_name = DeclareSort('Lifeline_name')
a = Const('a', Lifeline_name)
b = Const('b', Lifeline_name)
```

```
Lifeline_name = Function('Lifeline_name', l, Lifeline_name, BoolSort())
#--------------------------------
#lifelines_classes declarations
#--------------------------------
Lifeline_class = DeclareSort('Lifeline_class')
A = Const('A', Lifeline_class)
B = Const('B', Lifeline_class)
Lifeline_class = Function('Lifeline_class', l, Lifeline_class, BoolSort())
#--------------------------------
#Messages declarations
#--------------------------------
M  = DeclareSort('M')

Sd1_i = Const('Sd1_i', M)
Sd1_M1 = Const('Sd1_M1', M)
Sd1_j = Const('Sd1_j', M)
Sd1_M2 = Const('Sd1_M2', M)
Sd1_M31 = Const('Sd1_M31', M)
Sd1_M32 = Const('Sd1_M32', M)
M_i = Const('M_i', M)

Message_name = DeclareSort('Message_name')
m1 = Const('m1', Message_name)
m2 = Const('m2', Message_name)
i = Const('i', Message_name)
j = Const('j', Message_name)
m3 = Const('m3', Message_name)
Message_name = Function('Message_name', M, Message_name, BoolSort())
#--------------------------------
#Events declarations
#--------------------------------
Event = DeclareSort('Event')

Sd1_e2 = Const('Sd1_e2', Event)
Sd1_e3 = Const('Sd1_e3', Event)
Sd1_e6 = Const('Sd1_e6', Event)
Sd1_e7 = Const('Sd1_e7', Event)
Sd1_e91 = Const('Sd1_e91', Event)
Sd1_e92 = Const('Sd1_e92', Event)
#================================
Sd1_g2 = Const('Sd1_g2', Event)
Sd1_g3 = Const('Sd1_g3', Event)
```

```
Sd1_g6 = Const('Sd1_g6', Event)
Sd1_g7 = Const('Sd1_g7', Event)
Sd1_g91 = Const('Sd1_g91', Event)
Sd1_g92 = Const('Sd1_g92', Event)
#--------------------------------
#Declarations for the axioms
#--------------------------------
e_i = Const('e_i', Event)
e_j = Const('e_j', Event)
e_n = Const('e_n', Event)
#--------------------------------
#--------------------------------
#Elements Distinct
#--------------------------------
s = Solver()
e1 = Distinct( Sd1_e2, Sd1_g2, Sd1_e3, Sd1_g3,Sd1_e6,Sd1_e7,Sd1_e91,Sd1_e92, Sd1_g6,  Sd1_g7,
    Sd1_g91,Sd1_g92)
M1 = Distinct(Sd1_i,Sd1_j,Sd1_M31,Sd1_M32,Sd1_M2,Sd1_M1)
s.add(e1, M1)
#--------------------------------
#LES constraint,
#immediate Causality
#--------------------------------
iMNext = Function('iMNext', Event, Event, BoolSort())
s.add(ForAll ([e_i],(Not(iMNext(e_i, e_i)))))

#--------------------------------
##Normal Causality ->*
#--------------------------------
Next = Function('Next1', Event, Event, BoolSort())
s.add(ForAll ([e_i],(Next1(e_i, e_i))))
s.add(ForAll ([e_i,e_j], Implies(And(Next1(e_i, e_j),(e_i != e_j)),Not(Next1(e_j, e_i)))))
s.add(ForAll ([e_i,e_j,e_n], Implies(And(Next1(e_i, e_j),Next1(e_j, e_n)),(Next1(e_i, e_n)))))
# All events connected by immediate Causality(iMNext) are connected by Causality relation (
    Next)
s.add(ForAll ([e_i,e_j], Implies (iMNext(e_i,e_j) ,Next1(e_i,e_j))))
#--------------------------------
#immediate Conflict
#--------------------------------
iConflict = Function('iConflict', Event, Event, BoolSort())
#--------------------------------
#Normanl Conflict
#--------------------------------
```

```
Conflict = Function('Conflict', Event, Event, BoolSort())

s.add(ForAll ([e_i],(Not(Conflict(e_i, e_i)))))
s.add(ForAll ([e_i,e_j], Implies(Conflict(e_i, e_j),Conflict(e_j, e_i))))
s.add(ForAll ([e_i,e_j,e_n], Implies(And(And(Conflict(e_i, e_j),Next1(e_j, e_n))),(Conflict(
    e_i, e_n)))))
# All events connected by immediate conflict(iConflict) are connected by conflict relation (
    Conflict)
s.add(ForAll ([e_i,e_j], Implies (iConflict(e_i, e_j) ,Conflict(e_i, e_j))))
#--------------------------------
#Concurrency
#--------------------------------
Conc = Function('Conc', Event, Event, BoolSort())

s.add(ForAll([e_i, e_j],Conc(e_i, e_j)== Not(Or(Next1(e_i, e_j),Next1(e_j, e_i),Conflict(e_i,
    e_j)))))
#--------------------------------
#messages constraint, the messages has only one event as receive and the event connect to only
     one messages
#--------------------------------
isMsg = Function('isMsg',Event, M, Event, BoolSort())

s.add(ForAll([M_i, e_i],(Not(isMsg(e_i,M_i,e_i)))))
s.add(ForAll([e_i,M_i,e_j],Implies(isMsg(e_i,M_i,e_j),iMNext(e_i,e_j))))
#--------------------------------
cover = Function('cover', l, Event, BoolSort())

s.add(ForAll([L_i, e_i, L_j], Implies(And (cover(L_i, e_i),(L_i != L_j)), (Not(cover(L_j, e_i)
    )))))
#--------------------------------
# Sets of events of lifeline a
#--------------------------------
s.add(iMNext(Sd1_e2,Sd1_e6))
s.add(iMNext(Sd1_e2,Sd1_e7))
s.add(iMNext(Sd1_e3,Sd1_e6))
s.add(iMNext(Sd1_e3,Sd1_e7))
s.add(iMNext(Sd1_e6,Sd1_e91))
s.add(iMNext(Sd1_e7,Sd1_e92))
#--------------------------------
# Sets of events of lifeline b
#--------------------------------
s.add(iMNext(Sd1_g2,Sd1_g6))
s.add(iMNext(Sd1_g2,Sd1_g7))
```

```
s.add(iMNext(Sd1_g3,Sd1_g6))
s.add(iMNext(Sd1_g3,Sd1_g7))
s.add(iMNext(Sd1_g6,Sd1_g91))
s.add(iMNext(Sd1_g7,Sd1_g92))
#--------------------------------
# Sets of events in conflict
#--------------------------------
s.add(iConflict(Sd1_e6,Sd1_e7))
s.add(iConflict(Sd1_g6,Sd1_g7))
#--------------------------------
# Connect messages to send and receive events
#--------------------------------
s.add(isMsg(Sd1_e2,Sd1_i,Sd1_g2))
s.add(isMsg(Sd1_e3,Sd1_M1,Sd1_g3))
s.add(isMsg(Sd1_e6,Sd1_j,Sd1_g6))
s.add(isMsg(Sd1_e7,Sd1_M2,Sd1_g7))
s.add(isMsg(Sd1_e91,Sd1_M31,Sd1_g91))
s.add(isMsg(Sd1_e92,Sd1_M32,Sd1_g92))
#--------------------------------
# Connect Lifelines their events
#--------------------------------
s.add(cover(Sd1_a,Sd1_e2))
s.add(cover(Sd1_a,Sd1_e3))
s.add(cover(Sd1_a,Sd1_e6))
s.add(cover(Sd1_a,Sd1_e7))
s.add(cover(Sd1_a,Sd1_e91))
s.add(cover(Sd1_b,Sd1_g2))
s.add(cover(Sd1_b,Sd1_g3))
s.add(cover(Sd1_b,Sd1_g6))
s.add(cover(Sd1_b,Sd1_g7))
s.add(cover(Sd1_b,Sd1_g91))
#--------------------------------
# Connect message's to Message_names
#--------------------------------
Sd1_M1_name = Message_name(Sd1_i,i)
Sd1_M2_name = Message_name(Sd1_M1,m1)
Sd1_M3_name = Message_name(Sd1_j,j)
Sd1_M4_name = Message_name(Sd1_M2,m2)
Sd1_M5_name = Message_name(Sd1_M31,m3)
Sd1_M52_name = Message_name(Sd1_M32,m3)
#--------------------------------
# Connect lifeline's to lifeline_names
#--------------------------------
```

```
Sd1_L1_name = Lifeline_name(Sd1_a,a)
Sd1_L2_name = Lifeline_name(Sd1_b,b)
#--------------------------------
# Connect lifeline's to lifeline_classes
#--------------------------------
Sd1_L1_class = Lifeline_class(Sd1_a,A)
Sd1_L2_class =Lifeline_class(Sd1_b,B)
print s.check()
```

C.2 Z3 code for Sequence Diagram (sd2)

```
from z3 import *
#--------------------------------
#lifelines declarations
#--------------------------------
l = DeclareSort('l')
Sd1_a = Const('Sd1_a', l)
Sd1_b = Const('Sd1_b', l)
L_i = Const('L_i', l)
L_j = Const('L_j', l)
L_n = Const('L_n', l)
L_k = Const('L_k', l)
#--------------------------------
#Lifeline_name declarations
#--------------------------------
Lifeline_name = DeclareSort('Lifeline_name')
a = Const('a', Lifeline_name)
b = Const('b', Lifeline_name)
Lifeline_name = Function('Lifeline_name', l, Lifeline_name, BoolSort())
#--------------------------------
#lifelines_classes declarations
#--------------------------------
Lifeline_class = DeclareSort('Lifeline_class')
A = Const('A', Lifeline_class)
B = Const('B', Lifeline_class)
Lifeline_class = Function('Lifeline_class', l, Lifeline_class, BoolSort())
#--------------------------------
#Messages declarations
#--------------------------------
```

```
M  = DeclareSort('M')
M_j = Const('M_j', M)
Sd2_M1 = Const('Sd2_M1', M)
Sd2_new = Const('Sd2_new', M)
Sd2_M2 = Const('Sd2_M2', M)
Sd2_M4 = Const('Sd2_M4', M)
Sd2_M5 = Const('Sd2_M5', M)
#--------------------------------
#Events declarations
#--------------------------------
Event = DeclareSort('Event')

Sd2_f1 = Const('Sd2_f1', Event)
Sd2_f2 = Const('Sd2_f2', Event)
Sd2_f3 = Const('Sd2_f3', Event)
Sd2_f5 = Const('Sd2_f5', Event)
Sd2_f6 = Const('Sd2_f6', Event)
#================================
Sd2_h1 = Const('Sd2_h1', Event)
Sd2_h2 = Const('Sd2_h2', Event)
Sd2_h3 = Const('Sd2_h3', Event)
Sd2_h5 = Const('Sd2_h5', Event)
Sd2_h6 = Const('Sd2_h6', Event)
#--------------------------------
#Declarations for the axioms
#--------------------------------
g_i = Const('g_i', Event)
g_j = Const('g_j', Event)
g_n = Const('g_n', Event)
#--------------------------------
#Elements Distinct
#--------------------------------
s = Solver()
e2 = Distinct(Sd2_f1, Sd2_h1, Sd2_f2, Sd2_h2, Sd2_f3, Sd2_h3, Sd2_f5,Sd2_f6, Sd2_h5, Sd2_h6)
s.add(e2)
#--------------------------------
#LES constraint,
#immediate Causality
#--------------------------------
iMNext = Function('iMNext', Event, Event, BoolSort())
s.add(ForAll ([g_i],(Not(iMNext(g_i, g_i)))))

#--------------------------------
```

```
#Normal Causality ->*
#--------------------------------
Next = Function('Next', Event, Event, BoolSort())
s.add(ForAll ([g_i],(Next(g_i, g_i))))

s.add(ForAll ([g_i,g_j], Implies(And(Next(g_i, g_j),(g_i != g_j)),Not(Next(g_j, g_i)))))

s.add(ForAll ([g_i,g_j,g_n], Implies(And(Next(g_i, g_j),Next(g_j, g_n)),(Next(g_i, g_n)))))

# All events connected by immediate Causality(iMNext) are connected by Causality relation (
    Next)

s.add(ForAll ([g_i,g_j], Implies (iMNext(g_i,g_j) ,Next(g_i,g_j))))

#=========================
#immediate Conflict
#=========================
iConflict = Function('iConflict', Event, Event, BoolSort())
#--------------------------------
Conflict = Function('Conflict', Event, Event, BoolSort())

s.add(ForAll ([g_i],(Not(Conflict(g_i, g_i)))))
s.add(ForAll ([g_i,g_j], Implies(And(Conflict(g_i, g_j),(g_i != g_j)),Conflict(g_j, g_i))))
s.add(ForAll ([g_i,g_j,g_n], Implies(And(And(Conflict(g_i, g_j),Next(g_j, g_n))),(Conflict(g_i
    , g_n)))))
# All events connected by immediate conflict(iConflict) are connected by conflict relation (
    Conflict)
s.add(ForAll ([g_i,g_j], Implies (iConflict(g_i, g_j) ,Conflict(g_i, g_j))))
#--------------------------------
#Concurrency
#--------------------------------
Conc = Function('Conc', Event, Event, BoolSort())
s.add(ForAll([g_i, g_j],Conc(g_i, g_j)== Not(Or(Next(g_i, g_j),Next(g_j, g_i),Conflict(g_i,
    g_j)))))
#--------------------------------
#messages constraint, the messages has only one event as receive and the event connect to only
     one messages
#--------------------------------
isMsg = Function('isMsg',Event, M, Event, BoolSort())
s.add(ForAll([M_j, g_i],(Not(isMsg(g_i,M_j,g_i)))))
s.add(ForAll([g_i,M_j,g_j],Implies(isMsg(g_i,M_j,g_j),iMNext(g_i,g_j))))

cover = Function('cover', l, Event, BoolSort())
```

```
s.add(ForAll([L_n, g_i, L_k], Implies(And (cover(L_n, g_i),(L_n != L_k)), (Not(cover(L_k, g_i)
    )))))

#--------------------------------
# Sets of events of lifeline a
#--------------------------------
s.add(iMNext(Sd2_f1,Sd2_f2))
s.add(iMNext(Sd2_f2,Sd2_f3))
s.add(iMNext(Sd2_f3,Sd2_f5))
s.add(iMNext(Sd2_f3,Sd2_f6))
#--------------------------------
# Sets of events of lifeline b
#--------------------------------
s.add(iMNext(Sd2_h1,Sd2_h2))
s.add(iMNext(Sd2_h2,Sd2_h3))
s.add(iMNext(Sd2_h3,Sd2_h5))
s.add(iMNext(Sd2_h3,Sd2_h6))
#--------------------------------
# Sets of events in conflict
#--------------------------------
s.add(iConflict(Sd2_f5,Sd2_f6))
s.add(iConflict(Sd2_h5,Sd2_h6))
#--------------------------------
# Connect messages to send and receive events
#--------------------------------
s.add(isMsg(Sd2_f1,Sd2_M1,Sd2_h1))
s.add(isMsg(Sd2_h2,Sd2_new,Sd2_f2))
s.add(isMsg(Sd2_f3,Sd2_M2,Sd2_h3))
s.add(isMsg(Sd2_f5,Sd2_M4,Sd2_h5))
s.add(isMsg(Sd2_f6,Sd2_M5,Sd2_h6))
#--------------------------------
# Connect Lifelines their events
#--------------------------------
s.add(cover(Sd2_a,Sd2_f1))
s.add(cover(Sd2_a,Sd2_f2))
s.add(cover(Sd2_a,Sd2_f3))
s.add(cover(Sd2_a,Sd2_f5))
s.add(cover(Sd2_a,Sd2_f6))
s.add(cover(Sd2_b,Sd2_h1))
s.add(cover(Sd2_b,Sd2_h2))
s.add(cover(Sd2_b,Sd2_h3))
s.add(cover(Sd2_b,Sd2_h5))
s.add(cover(Sd2_b,Sd2_h6))
```

```
#--------------------------------
# Connect messages to Message_names
#--------------------------------
Sd2_M1_name = Message_name(Sd2_M1,m1)
Sd2_M2_name = Message_name(Sd2_M2,m2)
Sd2_M4_name = Message_name(Sd2_M4,m4)
Sd2_M5_name = Message_name(Sd2_M5,m5)
#--------------------------------
# Connect lifelines to lifeline_names
#--------------------------------
Sd2_a_name = Lifeline_name(Sd1_a,a)
Sd2_b_name = Lifeline_name(Sd1_b,b)
#--------------------------------
# Connect lifeline's to lifeline_classes
#--------------------------------
Sd2_a_class = Lifeline_class(Sd1_a,A)
Sd2_b_class =Lifeline_class(Sd1_b,B)
print s.check()
```

C.3 Z3 code for the composition of Sd1 and Sd2 (Sd3)

```
from z3 import *
#--------------------------------
#lifelines declarations of sd1
#--------------------------------
Lifeline1 = DeclareSort('Lifeline1')
Sd1_a = Const('Sd1_a', Lifeline1)
Sd1_b = Const('Sd1_b', Lifeline1)
#--------------------------------
#Declarations for the axioms
#--------------------------------
L_i = Const('L_i', Lifeline1)
L_j = Const('L_j', Lifeline1)
empty3 = Const('empty3', Lifeline1)
#--------------------------------
#lifelines declarations of sd2
#--------------------------------
Lifeline2 = DeclareSort('Lifeline2')
Sd2_a = Const('Sd2_a', Lifeline2)
```

```
Sd2_b = Const('Sd2_b', Lifeline2)
#--------------------------------
#Declarations for the axioms
#--------------------------------
L_n = Const('L_n', Lifeline2)
L_k = Const('L_k', Lifeline2)
empty4 = Const('empty4', Lifeline2)
#--------------------------------
#Lifeline_name declarations
#--------------------------------
Lifeline_name = DeclareSort('Lifeline_name')
a = Const('a', Lifeline_name)
b = Const('b', Lifeline_name)

Lifeline_name1 = Function('Lifeline_name', Lifeline1, Lifeline_name, BoolSort())
Lifeline_name2 = Function('Lifeline_name', Lifeline2, Lifeline_name, BoolSort())
#--------------------------------
#lifelines_classes declarations
#--------------------------------
Lifeline_class = DeclareSort('Lifeline_class')
A = Const('A', Lifeline_class)
B = Const('B', Lifeline_class)

Lifeline_class1 = Function('Lifeline_class1', Lifeline1, Lifeline_class, BoolSort())
Lifeline_class2 = Function('Lifeline_class2', Lifeline2, Lifeline_class, BoolSort())
#--------------------------------
#Messages declarations of sd1
#--------------------------------
Message1  = DeclareSort('Message1')
Sd1_i = Const('Sd1_i', Message1)
Sd1_M1 = Const('Sd1_M1', Message1)
Sd1_j = Const('Sd1_j', Message1)
Sd1_M2 = Const('Sd1_M2', Message1)
Sd1_M31 = Const('Sd1_M31', Message1)
Sd1_M32 = Const('Sd1_M32', Message1)
#--------------------------------
#Declarations for the axioms
#--------------------------------
M_i = Const('M_i', Message1)
#M_j = Const('M_j', Message1)
empty5 = Const('empty5', Message1)
#--------------------------------
#Messages declarations of sd2
```

```
#--------------------------------
Message2  = DeclareSort('Message2')
Sd2_M1 = Const('Sd2_M1', Message2)
Sd2_new = Const('Sd2_new', Message2)
Sd2_M2 = Const('Sd2_M2', Message2)
Sd2_M4 = Const('Sd2_M4', Message2)
Sd2_M5 = Const('Sd2_M5', Message2)
#--------------------------------
#Declarations for the axioms
#--------------------------------
M_j = Const('M_j', Message2)
empty6 = Const('empty6', Message2)
#--------------------------------
#Message_name declarations
#--------------------------------
Message_name = DeclareSort('Message_name')
m1 = Const('m1', Message_name)
new = Const('new', Message_name)
m2 = Const('m2', Message_name)
m4 = Const('m4', Message_name)
m5 = Const('m5', Message_name)
i = Const('i', Message_name)
j = Const('j', Message_name)
m3 = Const('m3', Message_name)
Message_name1 = Function('Message_name1', Message1, Message_name, BoolSort())
Message_name2 = Function('Message_name2', Message2, Message_name, BoolSort())


#--------------------------------
#Events declarations of sd1
#--------------------------------
Event1 = DeclareSort('Event1')
Sd1_e2 = Const('Sd1_e2', Event1)
Sd1_e3 = Const('Sd1_e3', Event1)
Sd1_e6 = Const('Sd1_e6', Event1)
Sd1_e7 = Const('Sd1_e7', Event1)
Sd1_e91 = Const('Sd1_e91', Event1)
Sd1_e92 = Const('Sd1_e92', Event1)
#================================
Sd1_g2 = Const('Sd1_g2', Event1)
Sd1_g3 = Const('Sd1_g3', Event1)
Sd1_g6 = Const('Sd1_g6', Event1)
Sd1_g7 = Const('Sd1_g7', Event1)
Sd1_g91 = Const('Sd1_g91', Event1)
```

```
Sd1_g92 = Const('Sd1_g92', Event1)
#--------------------------------
#Declarations for the axioms
#--------------------------------
empty1 = Const('empty1', Event1)
e_i = Const('e_i', Event1)
e_j = Const('e_j', Event1)
e_n = Const('e_n', Event1)
#--------------------------------
#Events declarations of sd2
#--------------------------------
Event2 = DeclareSort('Event2')
Sd2_f1 = Const('Sd2_f1', Event2)
Sd2_f2 = Const('Sd2_f2', Event2)
Sd2_f3 = Const('Sd2_f3', Event2)
Sd2_f5 = Const('Sd2_f5', Event2)
Sd2_f6 = Const('Sd2_f6', Event2)
#================================
Sd2_h1 = Const('Sd2_h1', Event2)
Sd2_h2 = Const('Sd2_h2', Event2)
Sd2_h3 = Const('Sd2_h3', Event2)
Sd2_h5 = Const('Sd2_h5', Event2)
Sd2_h6 = Const('Sd2_h6', Event2)
#--------------------------------
#Declarations for the axioms
#--------------------------------
g_i = Const('g_i', Event2)
g_j = Const('g_j', Event2)
g_n = Const('g_n', Event2)
empty2 = Const('empty2', Event2)
#--------------------------------
#Elements Distinct
#--------------------------------
s = Solver()
e2 = Distinct(Sd2_f1, Sd2_h1, Sd2_f2, Sd2_h2, Sd2_f3, Sd2_h3, Sd2_f5,Sd2_f6, Sd2_h5, Sd2_h6)
e1 = Distinct(Sd1_e2, Sd1_g2, Sd1_e3, Sd1_g3,Sd1_e6,Sd1_e7,Sd1_e91,Sd1_e92, Sd1_g6,  Sd1_g7,
    Sd1_g91,Sd1_g92)
M2 = Distinct(Sd2_M1, Sd2_M2,Sd2_new,Sd2_M4,Sd2_M5)
M1 = Distinct(Sd1_i,Sd1_j,Sd1_M31,Sd1_M32,Sd1_M2,Sd1_M1)
s.add(e1,e2, M1,M2)
#--------------------------------

#Cartesian product
```

```
#--------------------------------
list_e = [Sd1_e2, Sd1_g2, Sd1_e3, Sd1_g3,Sd1_e6,Sd1_e7,Sd1_e91,Sd1_e92, Sd1_g6,  Sd1_g7,
    Sd1_g91,Sd1_g92]
list_g = [Sd2_f1, Sd2_h1, Sd2_f2, Sd2_h2, Sd2_f3, Sd2_h3, Sd2_f5,Sd2_f6, Sd2_h5, Sd2_h6]

def mklist(l1, l2):
result = []
for x in l1:
for y in l2:
result.append((x,y))
return result
def addPairs(s, relation, relation2,empty1, empty2, l1, l2, matches):
L = {}
R = {}
for (x, y) in matches:
L[str(x)] = True
R[str(y)] = True
for (x,y) in mklist(l1, l2):
if str((x,y)) in map(str, matches):
s.add(relation(x,y))
s.add(relation2(x,y))

else:
s.add(Not(relation(x,y)))
for x in l1:
if not str(x) in L:
s.add(relation(x, empty2))
s.add(Notmatch1(x))
else:
s.add(Not(Notmatch1(x)))
for x in l2:
if not str(x) in R:
s.add(relation(empty1, x))
s.add(Notmatch2(x))
else:
s.add(Not(Notmatch2(x)))

#================================================================
#MessagePresent and MessageMatch functions
#================================================================
MessagePresent = Function('MessagePresent', Message1,Message2, BoolSort())
MessageMatch = Function('MessageMatch ', Message1, Message2, BoolSort())
```

```
#=====================================================================
#EventMatch and present functions
#=====================================================================
EventMatch = Function('EventMatch ', Event1, Event2, BoolSort())
present = Function('present', Event1,Event2, BoolSort())
#=====================================================================
#LifelineMatch and LifelinePresent functions
#=====================================================================
LifelinePresent = Function('LifelinePresent', Lifeline1,Lifeline2, BoolSort())
LifelineMatch = Function('LifelineMatch', Lifeline1, Lifeline2, BoolSort())
#=====================================================================
#Notmatch  functions for Events of sd1 and sd2
#=====================================================================
Notmatch1 = Function('Notmatch1', Event1, BoolSort())
Notmatch2 = Function('Notmatch2', Event2, BoolSort())
#=====================================================================
#MessageNotmatch functions for Messages of sd1 and sd2
#=====================================================================
MessageNotmatch1 = Function('MessageNotmatch1', Message1, BoolSort())
MessageNotmatch2 = Function('MessageNotmatch2', Message2, BoolSort())
#=====================================================================
#NotMatchEventLifeline for Lifelines of sd1 and sd2
#=====================================================================
LifelineNotmatch1 = Function('LifelineNotmatch1', Lifeline1, BoolSort())
s.add(ForAll ([L_i,L_n], Implies(LifelineMatch(L_i,L_n),Not(LifelineNotmatch1(L_i)))))

LifelineNotmatch2 = Function('LifelineNotmatch2', Lifeline2, BoolSort())
s.add(ForAll ([L_i,L_n], Implies(LifelineMatch(L_i,L_n),Not(LifelineNotmatch2(L_n)))))
#--------------------------------
#LES constraint,
#immediate Causality of sd1
#--------------------------------
iMNext1 = Function('iMNext1', Event1, Event1, BoolSort())
s.add(ForAll ([e_i],(Not(iMNext1(e_i, e_i)))))
#--------------------------------
#Normal Causality ->* of sd1
#--------------------------------
Next1 = Function('Next1', Event1, Event1, BoolSort())
s.add(ForAll ([e_i],(Next1(e_i, e_i))))
s.add(ForAll ([e_i,e_j], Implies(And(Next1(e_i, e_j),(e_i != e_j)),Not(Next1(e_j, e_i)))))
s.add(ForAll ([e_i,e_j,e_n], Implies(And(Next1(e_i, e_j),Next1(e_j, e_n)),(Next1(e_i, e_n)))))
# All events connected by immediate Causality(iMNext) are connected by Causality relation (
    Next)
```

```
s.add(ForAll ([e_i,e_j], Implies (iMNext1(e_i,e_j) ,Next1(e_i,e_j))))
#=========================
#-------------------------------
#LES constraint,
#immediate Causality of sd2
#-------------------------------
iMNext2 = Function('iMNext2', Event2, Event2, BoolSort())
s.add(ForAll ([g_i],(Not(iMNext2(g_i, g_i)))))
#-------------------------------
#Normal Causality ->* of sd2
#-------------------------------
Next2 = Function('Next2', Event2, Event2, BoolSort())
s.add(ForAll ([g_i],(Next2(g_i, g_i))))
s.add(ForAll ([g_i,g_j], Implies(And(Next2(g_i, g_j),(g_i != g_j)),Not(Next2(g_j, g_i)))))
s.add(ForAll ([g_i,g_j,g_n], Implies(And(Next2(g_i, g_j),Next2(g_j, g_n)),(Next2(g_i, g_n)))))
# All events connected by immediate Causality(iMNext) are connected by Causality relation (
    Next)
s.add(ForAll ([g_i,g_j], Implies (iMNext2(g_i,g_j) ,Next2(g_i,g_j))))
#=============================================
#LES constraint,
#immediate Causality for the composition
#=============================================
iMNext3  = Function('iMNext3 ', Event1,Event2,Event1, Event2, BoolSort())
s.add(ForAll ([e_i,g_i],Not(iMNext3(e_i,g_i,e_i,g_i))))
#=============================================
#LES constraint,
#Normal Causality for the composition
#=============================================
next3  = Function('next3 ', Event1,Event2,Event1, Event2, BoolSort())
s.add(ForAll ([e_i,g_i],(next3(e_i,g_i,e_i,g_i))))
s.add(ForAll ([e_i,g_i,e_j,g_j], Implies (And(next3(e_i,g_i,e_j,g_j),(e_i != e_j),(g_i != g_j)
    ),Not(next3(e_j,g_j,e_i,g_j)))))
s.add(ForAll ([e_i,g_i,e_j,g_j,e_n,g_n], Implies(And(next3(e_i,g_i,e_j,g_j),next3(e_j,g_j,e_n,
    g_n)),(next3(e_i,g_i,e_n,g_n)))))
s.add(ForAll ([e_i,g_i,e_j,g_j], Implies (iMNext3(e_i,g_i,e_j,g_j) ,next3(e_i,g_i,e_j,g_j))))
#-------------------------------
#immediate Conflict for sd1
#-------------------------------
iConflict1 = Function('iConflict1', Event1, Event1, BoolSort())
#-------------------------------
#Normanl Conflict for sd1
#-------------------------------
Conflict1 = Function('Conflict1', Event1, Event1, BoolSort())
```

```
s.add(ForAll ([e_i],(Not(Conflict1(e_i, e_i)))))
s.add(ForAll ([e_i,e_j], Implies(Conflict1(e_i, e_j),Conflict1(e_j, e_i))))
s.add(ForAll ([e_i,e_j,e_n], Implies(And(And(Conflict1(e_i, e_j),Next1(e_j, e_n))),(Conflict1(
    e_i, e_n)))))
# All events connected by immediate conflict(iConflict) are connected by conflict relation (
    Conflict)
s.add(ForAll ([e_i,e_j], Implies (iConflict1(e_i, e_j) ,Conflict1(e_i, e_j))))
#---------------------------------
#immediate Conflict for sd2
#---------------------------------
iConflict2 = Function('iConflict2', Event2, Event2, BoolSort())
#---------------------------------
#Normanl Conflict for sd2
#---------------------------------
Conflict2 = Function('Conflict2', Event2, Event2, BoolSort())
s.add(ForAll ([g_i],(Not(Conflict2(g_i, g_i)))))
s.add(ForAll ([g_i,g_j], Implies(And(Conflict2(g_i, g_j),(g_i != g_j)),Conflict2(g_j, g_i))))
s.add(ForAll ([g_i,g_j,g_n], Implies(And(And(Conflict2(g_i, g_j),Next2(g_j, g_n))),(Conflict2(
    g_i, g_n)))))
# All events connected by immediate conflict(iConflict) are connected by conflict relation (
    Conflict)
s.add(ForAll ([g_i,g_j], Implies (iConflict2(g_i, g_j) ,Conflict2(g_i, g_j))))
#---------------------------------
#immediate Conflict for composition
#---------------------------------
iConflict3 = Function('iConflict3', Event1,Event2,Event1, Event2, BoolSort())
#---------------------------------
#Normanl Conflict for composition
#---------------------------------
Conflict3 = Function('Conflict3', Event1,Event2,Event1, Event2, BoolSort())
s.add(ForAll ([e_i,g_i],(Not(Conflict3(e_i,g_i,e_i,g_i)))))
s.add(ForAll ([e_i,g_i,e_j,g_j], Implies (And(Conflict3(e_i,g_i,e_j,g_j),(e_i != e_j),(g_i !=
    g_j)),Conflict3(e_j,g_j,e_i,g_j))))
s.add(ForAll ([e_i,g_i,e_j,g_j,e_n,g_n], Implies(And(Conflict3(e_i,g_i,e_j,g_j),next3(e_j,g_j,
    e_n,g_n)),(Conflict3(e_i,g_i,e_n,g_n)))))
s.add(ForAll ([e_i,g_i,e_j,g_j], Implies (iConflict3(e_i,g_i,e_j,g_j) ,Conflict3(e_i,g_i,e_j,
    g_j))))
#---------------------------------
#Concurrency
#---------------------------------
Conc1 = Function('Conc1', Event1, Event1, BoolSort())
s.add(ForAll([e_i, e_j],Conc1(e_i, e_j)== Not(Or(Next1(e_i, e_j),Next1(e_j, e_i),Conflict1(e_i
    , e_j)))))
```

```
Conc2 = Function('Conc2', Event2, Event2, BoolSort())
s.add(ForAll([g_i, g_j],Conc2(g_i, g_j)== Not(Or(Next2(g_i, g_j),Next2(g_j, g_i),Conflict2(g_i
    , g_j)))))

Conc3 = Function('Conc3', Event1,Event2,Event1, Event2, BoolSort())

s.add(ForAll([e_i,g_i,e_j,g_j],Conc3(e_i,g_i,e_j,g_j)== Not(Or(next3(e_i,g_i,e_j,g_j),next3(
    e_j,g_j,e_i,g_j),Conflict3(e_i,g_i,e_j,g_j)))))
#--------------------------------
#messages constraint, the messages has only one event as receive and the event connect to only
     one messages
#--------------------------------
isMsg2 = Function('isMsg2',Event2, Message2, Event2, BoolSort())
s.add(ForAll([M_j, g_i],(Not(isMsg2(g_i,M_j,g_i)))))
s.add(ForAll([g_i,M_j,g_j],Implies(isMsg2(g_i,M_j,g_j),iMNext2(g_i,g_j))))
s.add(ForAll([g_i,M_j,g_j],Implies(And(And(isMsg2(g_i,M_j,g_j),iMNext2(g_i,g_j)),Notmatch2(g_i
    ),Notmatch2(g_j)),iMNext3(empty1,g_i,empty1,g_j))))
#--------------------------------
#messages constraint, the messages has only one event as receive and the event connect to only
     one messages
#--------------------------------
isMsg1 = Function('isMsg1',Event1, Message1, Event1, BoolSort())
s.add(ForAll([M_i, e_i],(Not(isMsg1(e_i,M_i,e_i)))))
s.add(ForAll([e_i,M_i,e_j],Implies(isMsg1(e_i,M_i,e_j),iMNext1(e_i,e_j))))
s.add(ForAll([e_i,M_i,e_j],Implies(And(And(isMsg1(e_i,M_i,e_j),iMNext1(e_i,e_j)),Notmatch1(e_i
    ),Notmatch1(e_j)),iMNext3(e_i,empty2,e_j,empty2))))
#--------------------------------
#Relation isMsg3 for composition
#--------------------------------
isMsg3 = Function('isMsg3',Event1,Event2, Message1,Message2, Event1,Event2, BoolSort())
#--------------------------------
#Matching axioms for messages
#--------------------------------
s.add(ForAll ([e_i,g_i,e_j,g_j,M_i,M_j], Implies

(And(And(MessageNotmatch1(M_i),MessageNotmatch2(M_j)),isMsg1(e_i,M_i,e_j),isMsg2(g_i,M_j,g_j))
    ,
And(isMsg3(e_i,empty2,M_i,empty6,e_j,empty2),isMsg3(empty1,g_i,empty5,M_j,empty1,g_j)))))

s.add(ForAll ([e_i,e_j,g_i,g_j,M_i,M_j], Implies (And(EventMatch(e_i,g_i),isMsg1(e_i,M_i,e_j),
    isMsg2(g_i,M_j,g_j)),MessageMatch (M_i, M_j))))
```

```
s.add(ForAll ([e_i,e_j,g_i,g_j,M_i,M_j], Implies (And(EventMatch(e_j,g_j),isMsg1(e_i,M_i,e_j),
    isMsg2(g_i,M_j,g_j)),MessageMatch (M_i, M_j))))

s.add(ForAll ([e_i,e_j,g_i,g_j,M_i,M_j], Implies (And(MessageMatch (M_i, M_j),isMsg1(e_i,M_i,
    e_j),isMsg2(g_i,M_j,g_j)),EventMatch(e_i,g_i))))
s.add(ForAll ([e_i,e_j,g_i,g_j,M_i,M_j], Implies (And(MessageMatch (M_i, M_j),isMsg1(e_i,M_i,
    e_j),isMsg2(g_i,M_j,g_j)),EventMatch (e_j, g_j))))

s.add(ForAll ([e_i,g_i,e_j,g_j,M_i,M_j], Implies

(And(And(MessageMatch (M_i, M_j)),isMsg1(e_i,M_i,e_j),isMsg2(g_i,M_j,g_j)),isMsg3(e_i,g_i,M_i,
    M_j,e_j,g_j))))
#---------------------------------
#cover relation for sd1
#---------------------------------
cover1 = Function('cover1', Lifeline1, Event1, BoolSort())
s.add(ForAll([L_i, e_i, L_j], Implies(And (cover1(L_i, e_i),(L_i != L_j)), (Not(cover1(L_j,
    e_i))))))
#---------------------------------
#cover relation for sd2
#---------------------------------
cover2 = Function('cover2', Lifeline2, Event2, BoolSort())
s.add(ForAll([L_n, g_i, L_k], Implies(And (cover2(L_n, g_i),(L_n != L_k)), (Not(cover2(L_k,
    g_i))))))
#=========================
#cover relation for the composition
#=========================
cover3 = Function('cover3', Lifeline1,Lifeline2, Event1,Event2, BoolSort())
#---------------------------------
#---------------------------------
#Matching axioms for lifelines
#---------------------------------
s.add(ForAll ([e_i,g_i,L_i,L_n], Implies (And(And(LifelineMatch(L_i, L_n),EventMatch(e_i,g_i))
    ,cover1(L_i,e_i),cover2(L_n,g_i)),cover3(L_i,L_n,e_i,g_i))))
s.add(ForAll ([e_i,g_i,L_i,L_n], Implies (And(And(And(cover1(L_i,e_i),cover2(L_n,g_i)),
    Notmatch1(e_i),Notmatch2(g_i)),LifelineNotmatch2(L_n),LifelineNotmatch1(L_i)),And(cover3(
    L_i,empty4,e_i,empty2),cover3(empty3,L_n,empty1,g_i)))))
s.add(ForAll ([e_i,L_i,L_n], Implies (And(LifelineMatch(L_i, L_n),cover1(L_i,e_i), Notmatch1(
    e_i)),cover3(L_i,L_n,e_i,empty2))))
s.add(ForAll ([g_i,L_i,L_n], Implies (And(LifelineMatch(L_i, L_n),cover2(L_n,g_i), Notmatch2(
    g_i)),cover3(L_i,L_n,empty1,g_i))))
s.add(ForAll ([e_i,g_i,L_i,L_n], Implies (And(And(EventMatch(e_i,g_i),cover1(L_i,e_i),cover2(
    L_n,g_i))),LifelineMatch(L_i,L_n))))
```

```
#--------------------------------
#---------------------------------------------------------
#Matching axioms for Events in Causality relation
#---------------------------------------------------------
s.add(ForAll ([e_i,g_i,e_j,g_j], Implies(And(And(iMNext1(e_i, e_j),iMNext2(g_i,g_j)),
    EventMatch(e_i, g_i),EventMatch(e_j, g_j)),iMNext3(e_i,g_i,e_j,g_j))))
s.add(ForAll ([e_i,g_i,e_j,g_j], Implies (And(And(And(iMNext1(e_i, e_j),iMNext2(g_i,g_j)),
    Notmatch1(e_i),Notmatch2(g_i)),Notmatch1(e_j),Notmatch2(g_j)),And(iMNext3(e_i,empty2,e_j,
    empty2),iMNext3(empty1,g_i,empty1,g_j)))))
s.add(ForAll ([e_i,g_i,e_j], Implies(And(And(EventMatch(e_i,g_i),iMNext1(e_i,e_j)),Notmatch1(
    e_j)),iMNext3(e_i,g_i,e_j,empty2))))
s.add(ForAll ([e_i,g_i,g_j], Implies(And(And(EventMatch(e_i,g_i),iMNext2(g_i,g_j)),Notmatch2(
    g_j)),iMNext3(e_i,g_i,empty1,g_j))))
s.add(ForAll ([e_i,g_j,e_j], Implies(And(And(EventMatch(e_j, g_j),iMNext1(e_i,e_j)),Notmatch1(
    e_i)),iMNext3(e_i,empty2,e_j,g_j))))
s.add(ForAll ([e_j,g_i,g_j], Implies(And(And(EventMatch(e_j, g_j),iMNext2(g_i,g_j)),Notmatch2(
    g_i)),iMNext3(empty1,g_i,e_j,g_j))))
#---------------------------------------------------------
#Matching axioms for Events in iConflict1 relation
#---------------------------------------------------------
s.add(ForAll ([e_i,g_i,e_j,g_j], Implies(And(And(iConflict1(e_i, e_j),iConflict2(g_i,g_j)),
    EventMatch(e_i, g_i),EventMatch(e_j, g_j)),iConflict3(e_i,g_i,e_j,g_j))))
s.add(ForAll ([e_i,g_i,e_j], Implies(And(And(EventMatch(e_i,g_i),iConflict1(e_i,e_j)),
    Notmatch1(e_j)),iConflict3(e_i,g_i,e_j,empty2))))
s.add(ForAll ([e_i,g_i,g_j], Implies(And(And(EventMatch(e_i,g_i),iConflict2(g_i,g_j)),
    Notmatch2(g_j)),iConflict3(e_i,g_i,empty1,g_j))))
s.add(ForAll ([e_i,g_j,e_j], Implies(And(And(EventMatch(e_j, g_j),iConflict1(e_i,e_j)),
    Notmatch1(e_i)),iConflict3(e_i,empty2,e_j,g_j))))
s.add(ForAll ([e_j,g_i,g_j], Implies(And(And(EventMatch(e_j, g_j),iConflict2(g_i,g_j)),
    Notmatch2(g_i)),iConflict3(empty1,g_i,e_j,g_j))))
#--------------------------------
#--------------------------------
#general order between events of sd2
s.add(iMNext2(Sd2_f1,Sd2_f2))
s.add(iMNext2(Sd2_f2,Sd2_f3))
s.add(iMNext2(Sd2_f3,Sd2_f5))
s.add(iMNext2(Sd2_f3,Sd2_f6))
s.add(iMNext2(Sd2_h1,Sd2_h2))
s.add(iMNext2(Sd2_h2,Sd2_h3))
s.add(iMNext2(Sd2_h3,Sd2_h5))
s.add(iMNext2(Sd2_h3,Sd2_h6))
#Conflict2 between events of sd2
s.add(iConflict2(Sd2_f5,Sd2_f6))
```

```
s.add(iConflict2(Sd2_h5,Sd2_h6))
#=========================
#general order between events of sd1
s.add(iMNext1(Sd1_e2,Sd1_e6))
s.add(iMNext1(Sd1_e2,Sd1_e7))
s.add(iMNext1(Sd1_e3,Sd1_e6))
s.add(iMNext1(Sd1_e3,Sd1_e7))
s.add(iMNext1(Sd1_e6,Sd1_e91))
s.add(iMNext1(Sd1_e7,Sd1_e92))
s.add(iMNext1(Sd1_g2,Sd1_g6))
s.add(iMNext1(Sd1_g2,Sd1_g7))
s.add(iMNext1(Sd1_g3,Sd1_g6))
s.add(iMNext1(Sd1_g3,Sd1_g7))
s.add(iMNext1(Sd1_g6,Sd1_g91))
s.add(iMNext1(Sd1_g7,Sd1_g92))
#Conflict2 between events of sd1
s.add(iConflict1(Sd1_e6,Sd1_e7))
s.add(iConflict1(Sd1_g6,Sd1_g7))
#--------------------------------
# Connect messages to send and receive events of sd2
#--------------------------------
s.add(isMsg2(Sd2_f1,Sd2_M1,Sd2_h1))
s.add(isMsg2(Sd2_h2,Sd2_new,Sd2_f2))
s.add(isMsg2(Sd2_f3,Sd2_M2,Sd2_h3))
s.add(isMsg2(Sd2_f5,Sd2_M4,Sd2_h5))
s.add(isMsg2(Sd2_f6,Sd2_M5,Sd2_h6))
#--------------------------------
# Connect messages to send and receive events of sd1
#--------------------------------
s.add(isMsg1(Sd1_e2,Sd1_i,Sd1_g2))
s.add(isMsg1(Sd1_e3,Sd1_M1,Sd1_g3))
s.add(isMsg1(Sd1_e6,Sd1_j,Sd1_g6))
s.add(isMsg1(Sd1_e7,Sd1_M2,Sd1_g7))
s.add(isMsg1(Sd1_e91,Sd1_M31,Sd1_g91))
s.add(isMsg1(Sd1_e92,Sd1_M32,Sd1_g92))
#--------------------------------
# Connect Lifelines their events sd2
#--------------------------------
s.add(cover2(Sd2_a,Sd2_f1))
s.add(cover2(Sd2_a,Sd2_f2))
s.add(cover2(Sd2_a,Sd2_f3))
s.add(cover2(Sd2_a,Sd2_f5))
s.add(cover2(Sd2_a,Sd2_f6))
```

```
s.add(cover2(Sd2_b,Sd2_h1))
s.add(cover2(Sd2_b,Sd2_h2))
s.add(cover2(Sd2_b,Sd2_h3))
s.add(cover2(Sd2_b,Sd2_h5))
s.add(cover2(Sd2_b,Sd2_h6))
#--------------------------------
# Connect Lifelines their events sd1
#--------------------------------
s.add(cover1(Sd1_a,Sd1_e2))
s.add(cover1(Sd1_a,Sd1_e3))
s.add(cover1(Sd1_a,Sd1_e6))
s.add(cover1(Sd1_a,Sd1_e7))
s.add(cover1(Sd1_a,Sd1_e91))
s.add(cover1(Sd1_b,Sd1_g2))
s.add(cover1(Sd1_b,Sd1_g3))
s.add(cover1(Sd1_b,Sd1_g6))
s.add(cover1(Sd1_b,Sd1_g7))
s.add(cover1(Sd1_b,Sd1_g91))
#=========================
#--------------------------------
# Connect message's to Message_names
#--------------------------------
Sd2_M1_name = Message_name2(Sd2_M1,m1)
Sd2_M2_name = Message_name2(Sd2_new,new)
Sd2_M3_name = Message_name2(Sd2_M2,m2)
Sd2_M4_name = Message_name2(Sd2_M4,m4)
Sd2_M5_name = Message_name2(Sd2_M5,m5)

Sd1_M1_name = Message_name1(Sd1_i,i)
Sd1_M2_name = Message_name1(Sd1_M1,m1)
Sd1_M3_name = Message_name1(Sd1_j,j)
Sd1_M4_name = Message_name1(Sd1_M2,m2)
Sd1_M5_name = Message_name1(Sd1_M31,m3)
Sd1_M52_name = Message_name1(Sd1_M32,m3)

#--------------------------------
# Connect lifeline's to lifeline_names
#--------------------------------
Sd1_a_name = Lifeline_name1(Sd1_a,a)
Sd1_b_name = Lifeline_name1(Sd1_b,b)
Sd2_a_name = Lifeline_name2(Sd2_a,a)
Sd2_b_name = Lifeline_name2(Sd2_b,b)
#--------------------------------
```

```
# Connect lifeline's to lifeline_classes
#--------------------------------
Sd2_a_class = Lifeline_class2(Sd2_a,A)
Sd2_b_class = Lifeline_class2(Sd2_b,B)
Sd1_a_class =Lifeline_class1 (Sd1_a,A)
Sd1_b_class = Lifeline_class1(Sd1_b,B)

def NameCheck (*items):
for i in range(1, len(items)):
if (str(items[i].arg(1)) != str(items[i-1].arg(1))):
return False
return True

def main():
if NameCheck (Sd2_M1_name, Sd1_M2_name) == False:
print "Message1 not equals"
return
if NameCheck (Sd2_M3_name, Sd1_M4_name) == False:
print "Message2 not equals"
return
if NameCheck (Sd1_a_name, Sd2_a_name) == False:
print "Lifelines 1 names are not equals"
return
if NameCheck (Sd1_b_name, Sd2_b_name) == False:
print "Lifelines 2 names are equals"
return
if NameCheck (Sd2_a_class, Sd1_a_class) == False:
print "Lifelines 1 class are not equals"
return
if NameCheck (Sd2_b_class, Sd1_b_class) == False:
print "Lifelines 2 class are equals"
return
main()
cover_info_event_message = {}
messageMatches = {}
allMessage1 = {}
allMessage2 = {}
#--------------------------------
# Process Message matches
# ==== START
assertions = s.assertions()
for ast in assertions:
if not "is_forall" in dir(ast) and str(ast.decl()) in ["isMsg1", "isMsg2"]:
```

```
cover_info_event_message[str(ast.arg(0))] = ast.arg(1)
cover_info_event_message[str(ast.arg(2))] = ast.arg(1)
if "1" in str(ast.arg(1).sort()):
allMessage1[str(ast.arg(1))] = ast.arg(1)
if "2" in str(ast.arg(1).sort()):
allMessage2[str(ast.arg(1))] = ast.arg(1)
for (x,y) in matches:
oldx = x
oldy = y
x1 = cover_info_event_message[str(oldx)]
y1 = cover_info_event_message[str(oldy)]

pair11 = (str(x1), str(y1))
pair12 = (str(y1), str(x1))
messageMatches[str(pair11)] = True
messageMatches[str(pair12)] = True
allMessagePairs = mklist(allMessage1, allMessage2)
messageMatched = {}
for (x, y) in allMessagePairs:
x = allMessage1[x]
y = allMessage2[y]
if str(x) == str(y): continue
if str((str(x), str(y))) in messageMatches:
s.add(MessageMatch(x, y))
s.add(MessagePresent(x, y))

messageMatched[str(x)]=True
messageMatched[str(y)]=True
else:
s.add(Not(MessagePresent(x, y)))

for l in allMessage1:
if not str(l) in messageMatched:
s.add(MessagePresent(allMessage1[l], empty6))
s.add(MessageNotmatch1(allMessage1[l]))
else:
s.add(Not(MessageNotmatch1(allMessage1[l])))
for l in allMessage2:
if not str(l) in messageMatched:
s.add(MessagePresent(empty5, allMessage2[l]))
s.add(MessageNotmatch2(allMessage2[l]))
else:
s.add(Not(MessageNotmatch2(allMessage2[l])))
```

```
# ==== END
cover_info_event_lineline = {}
lifelineMatches = {}
allLifelines1 = {}
allLifelines2 = {}
#--------------------------------
# Process lifeline matches
# ==== START
assertions = s.assertions()
for ast in assertions:
if not "is_forall" in dir(ast) and str(ast.decl()) in ["cover1", "cover2"]:
cover_info_event_lineline[str(ast.arg(1))] = ast.arg(0)
if "1" in str(ast.arg(0).sort()):
allLifelines1[str(ast.arg(0))] = ast.arg(0)
if "2" in str(ast.arg(0).sort()):
allLifelines2[str(ast.arg(0))] = ast.arg(0)
for (x,y) in matches:
oldx = x
oldy = y
x = cover_info_event_lineline[str(x)]
y = cover_info_event_lineline[str(y)]
pair1 = (str(x), str(y))
pair2 = (str(y), str(x))
lifelineMatches[str(pair1)] = True
lifelineMatches[str(pair2)] = True
allLifelinePairs = mklist(allLifelines1, allLifelines2)
lifelineMatched = {}
for (x, y) in allLifelinePairs:
x = allLifelines1[x]
y = allLifelines2[y]
if str(x) == str(y): continue
if str((str(x), str(y))) in lifelineMatches:
s.add(LifelineMatch(x, y))
s.add(LifelinePresent(x, y))
lifelineMatched[str(x)]=True
lifelineMatched[str(y)]=True
else:
s.add(Not(LifelineMatch(x, y)))
s.add(Not(LifelinePresent(x, y)))

for l in allLifelines1:
if not str(l) in lifelineMatched:
s.add(LifelineNotmatch1(allLifelines1[l]))
```

```
s.add(LifelinePresent(allLifelines1[l], empty4))

else:
s.add(Not(LifelineNotmatch1(allLifelines1[l])))
s.add(Not(LifelinePresent(allLifelines1[l], empty4)))

for l in allLifelines2:
if not str(l) in lifelineMatched:
s.add(LifelineNotmatch2(allLifelines2[l]))
s.add(LifelinePresent(empty3, allLifelines2[l]))

else:
s.add(Not(LifelineNotmatch2(allLifelines2[l])))
s.add(Not(LifelinePresent(empty3, allLifelines2[l])))

addPairs(s, present, EventMatch, empty1, empty2, list_e, list_g, matches)
matches = [
(Sd1_e3,Sd2_f1),(Sd1_g3,Sd2_h1),(Sd1_e7,Sd2_f3),(Sd1_g7,Sd2_h3)
]
print s.check()
```

C.4 Z3 code for the advice model of the petrol station example in section 6.4

```
from z3 import *
import os
#---------------------------------
#Lifeline declarations
#---------------------------------
Lifeline = DeclareSort('Lifeline')
User = Const('User', Lifeline)
PetrolStation = Const('PetrolStation', Lifeline)
Bank = Const('Bank', Lifeline)
L1 = Const('L1', Lifeline)
L2 = Const('L2', Lifeline)
#---------------------------------
#lifelines_classes declarations
#---------------------------------
```

```
Lifeline_class = DeclareSort('Lifeline_class')
class_User = Const('class_User', Lifeline_class)
class_PetrolStation = Const('class_PetrolStation', Lifeline_class)
class_Bank = Const('class_Bank', Lifeline)
Lifeline_class = Function('Lifeline_class', Lifeline, Lifeline_class, BoolSort())
#--------------------------------
#Events declarations
#--------------------------------
Event = DeclareSort('Event')
e1 = Const('e1', Event)
e2 = Const('e2', Event)
e3 = Const('e3', Event)
e5 = Const('e5', Event)
e6 = Const('e6', Event)
#================================
g1 = Const('g1', Event)
g2 = Const('g2', Event)
g3 = Const('g3', Event)
g4 = Const('g4', Event)
g6 = Const('g6', Event)
g7 = Const('g7', Event)
g8 = Const('g8', Event)
g9 = Const('g9', Event)
g10 = Const('g10', Event)
#================================
l1 = Const('l1', Event)
l3 = Const('l3', Event)
l4 = Const('l4', Event)
l5 = Const('l5', Event)
#--------------------------------
#Messages declarations
#--------------------------------
Message = DeclareSort('Message')
ValidPin = Const('ValidPin', Message)
EnterFuelAmount = Const('EnterFuelAmount', Message)
FuelAmount = Const('FuelAmount', Message)
StartFuel = Const('StartFuel', Message)
CheckAmount = Const('CheckAmount', Message)
BalanceOk = Const('BalanceOk', Message)
Withdrow = Const('Withdrow', Message)
Cancel = Const('Cancel', Message)
PaymentDeclined = Const('PaymentDeclined', Message)
m = Const('m', Message)
```

```
#--------------------------------
#Message_name declarations
#--------------------------------
Message_name = DeclareSort('Message_name')
Name_ValidPin = Const('Name_ValidPin', Message_name)
Name_EnterFuelAmount = Const('Name_EnterFuelAmount', Message_name)
Name_FuelAmount = Const('Name_FuelAmount', Message_name)
Name_StartFuel = Const('Name_StartFuel', Message_name)
Name_CheckAmount = Const('Name_CheckAmount', Message_name)
Name_BalanceOk = Const('Name_BalanceOk', Message_name)
Name_Withdrow = Const('Name_Withdrow', Message_name)
Name_Cancel = Const('Name_Cancel', Message_name)
Name_PaymentDeclined = Const('Name_PaymentDeclined', Message_name)
Name_Message_name = Function('Message_name', Message, Message_name, BoolSort())
#--------------------------------
#Distinct
#--------------------------------
s = Solver()
e = Distinct(e1, g1, e2, g2, e3, g3,e5,e6,g4, g6,  g7,g8,g9, g10, l1,  l3, l4, l5)
M = Distinct(FuelAmount,CheckAmount, Withdrow, ValidPin, EnterFuelAmount, StartFuel, Cancel,
    PaymentDeclined, BalanceOk)
L = Distinct(User, PetrolStation, Bank)
s.add(e,M,L)
#--------------------------------
#Events constraint,
#--------------------------------
iMNext = Function('iMNext', Event, Event, BoolSort())
s.add(ForAll ([e1],(Not(iMNext(e1, e1)))))
Next = Function('Next', Event, Event, BoolSort())
s.add(ForAll ([e1],(Next(e1, e1))))
s.add(ForAll ([e1,e2], Implies(And(Next(e1, e2),(e1 != e2)),Not(Next(e2, e1)))))
s.add(ForAll ([e1,e2,e3], Implies(And(And(Next(e1, e2),Next(e2, e3))),(Next(e1, e3)))))
s.add(ForAll ([e1,e2], Implies (And(iMNext(e1, e2),(e1 != e2)) ,Next(e1, e2))))
#--------------------------------
#Conflict Function
#--------------------------------
iConflict = Function('iConflict', Event, Event, BoolSort())
Conflict = Function('Conflict', Event, Event, BoolSort())
s.add(ForAll ([e1],(Not(Conflict(e1, e1)))))
s.add(ForAll ([e1,e2], Implies(And(Conflict(e1, e2),(e1 != e2)),Conflict(e2, e1))))
s.add(ForAll ([e1,e2,e3], Implies(And(And(Conflict(e1, e2),Next(e2, e3))),(Conflict(e1, e3))))
    )
s.add(ForAll ([e1,e2], Implies (And(iConflict(e1,e2),(e1 != e2)) ,Conflict(e1,e2))))
```

```
#--------------------------------
#Concurrency
#--------------------------------
Conc = Function('Conc', Event, Event, BoolSort())
s.add(ForAll([e1, e2],Conc(e1, e2) == Not(Or(Conflict(e1, e2),Next(e1, e2),Next(e2, e1)))))
#--------------------------------
#lifeline relation with events
#--------------------------------

cover = Function('cover', Lifeline, Event, BoolSort())
s.add(ForAll([L1, e1, L2], Implies(And (cover(L1, e1),(L1 != L2)), (Not(cover(L2, e1))))))
#--------------------------------
#Message  and its events relation
#--------------------------------
isMsg = Function ('isMsg', Event, Message, Event, BoolSort())
s.add(ForAll([e1,m,e2],Implies(isMsg(e1,m,e2),iMNext(e1,e2))))
s.add(ForAll([e1,m],(Not(isMsg(e1,m,e1)))))
#--------------------------------
#general order between events
#--------------------------------
s.add(iMNext(e1,e2))
s.add(iMNext(e2,e3))
s.add(iMNext(e3,e5))
s.add(iMNext(e3,e6))
#=========================
s.add(iMNext(g1,g2))
s.add(iMNext(g2,g3))
s.add(iMNext(g3,g4))
s.add(iMNext(g4,g6))
s.add(iMNext(g6,g7))
s.add(iMNext(g7,g8))
s.add(iMNext(g4,g9))
s.add(iMNext(g9,g10))

s.add(iConflict(e5,e6))
s.add(iConflict(g6,g9))
#=========================
s.add(iMNext(l1,l3))
s.add(iMNext(l3,l4))
s.add(iMNext(l1,l5))
s.add(iConflict(l3,l5))
#--------------------------------
# Connect message's to send event and receive event
```

```
#-------------------------------
s.add(isMsg(g1,ValidPin,e1))
s.add(isMsg(g2,EnterFuelAmount,e2))
s.add(isMsg(e3,FuelAmount,g3))
s.add(isMsg(g4,CheckAmount,l1))
s.add(isMsg(g8,StartFuel,e5))
s.add(isMsg(g10,PaymentDeclined,e6))
s.add(isMsg(l4,Withdrow,g7))
s.add(isMsg(l3,BalanceOk,g6))
s.add(isMsg(l5,Cancel,g9))
#==========================
#assigning lifeline with its events
#==========================
s.add(cover(User,e1))
s.add(cover(User,e2))
s.add(cover(User,e3))
s.add(cover(User,e5))
s.add(cover(User,e6))
s.add(cover(PetrolStation,g1))
s.add(cover(PetrolStation,g2))
s.add(cover(PetrolStation,g3))
s.add(cover(PetrolStation,g4))
s.add(cover(PetrolStation,g6))
s.add(cover(PetrolStation,g7))
s.add(cover(PetrolStation,g8))
s.add(cover(PetrolStation,g9))
s.add(cover(PetrolStation,g10))
s.add(cover(Bank,l1))
s.add(cover(Bank,l3))
s.add(cover(Bank,l4))
s.add(cover(Bank,l5))
#--------------------------------
#graph generator
#--------------------------------
print (s.check())
print (s.model())
```

C.5 Z3 code for the base model of the petrol station example in section 6.4

```
from z3 import *
#--------------------------------
#Lifeline declarations
#--------------------------------
Lifeline = DeclareSort('Lifeline')
User = Const('User', Lifeline)
PetrolStation = Const('PetrolStation', Lifeline)
Bank = Const('Bank', Lifeline)
L1 = Const('L1', Lifeline)
L2 = Const('L2', Lifeline)
#--------------------------------
#lifelines_classes declarations
#--------------------------------
Lifeline_class = DeclareSort('Lifeline_class')
class_User = Const('class_User', Lifeline_class)
class_PetrolStation = Const('class_PetrolStation', Lifeline_class)
class_Bank = Const('class_Bank', Lifeline)
Lifeline_class = Function('Lifeline_class', Lifeline, Lifeline_class, BoolSort())
#--------------------------------
#Events declarations
#--------------------------------
Event = DeclareSort('Event')
e1 = Const('e1', Event)
e2 = Const('e2', Event)
e3 = Const('e3', Event)
e5 = Const('e5', Event)
e6 = Const('e6', Event)
e7 = Const('e7', Event)
e8 = Const('e8', Event)
e101 = Const('e101', Event)
e102 = Const('e102', Event)
#================================
g1 = Const('g1', Event)
g2 = Const('g2', Event)
g3 = Const('g3', Event)
g4 = Const('g4', Event)
g5 = Const('g5', Event)
```

```
g7 = Const('g7', Event)
g8 = Const('g8', Event)
g9 = Const('g9', Event)
g10 = Const('g10', Event)
g121 = Const('g121', Event)
g122 = Const('g122', Event)
#================================
l1 = Const('l1', Event)
l2 = Const('l2', Event)
#================================
#Messages declarations
#================================
Message = DeclareSort('Message')
InserCard = Const('InserCard', Message)
RequestPin = Const('RequestPin', Message)
PinCode = Const('PinCode', Message)
Validate = Const('Validate', Message)
Result = Const('Result', Message)
ValidPin = Const('ValidPin', Message)
StartFuel = Const('StartFuel', Message)
Stop = Const('Stop', Message)
InvalidPin = Const('InvalidPin', Message)
CardOut = Const('CardOut', Message)
m = Const('m', Message)
#--------------------------------
#Message_name declarations
#--------------------------------
Message_name = DeclareSort('Message_name')
Name_InserCard = Const('Name_InserCard', Message_name)
Name_RequestPin = Const('Name_RequestPin', Message_name)
Name_PinCode = Const('Name_PinCode', Message_name)
Name_Validate = Const('Name_Validate', Message_name)
Name_Result = Const('Name_Result', Message_name)
Name_ValidPin = Const('Name_ValidPin', Message_name)
Name_StartFuel = Const('Name_StartFuel', Message_name)
Name_Stop = Const('Name_Stop', Message_name)
Name_InvalidPin = Const('Name_InvalidPin', Message_name)
Name_CardOut = Const('Name_CardOut', Message_name)
Name_Message_name = Function('Message_name', Message, Message_name, BoolSort())
#--------------------------------
#Distinct
#--------------------------------
s = Solver()
```

```
e = Distinct( e1, g1, e2, g2, e3, g3, g4, e5, g5,e6, e7, g7,e8,e101,e102,g8,g9,g10,g122,l1, l2
    )
M = Distinct(InserCard,RequestPin, PinCode, Validate,ValidPin, Result, StartFuel, Stop,
    InvalidPin, CardOut)
L = Distinct(User, PetrolStation, Bank)
s.add(e,M,L)
#--------------------------------
#Events constraint,
#--------------------------------
iMNext = Function('iMNext', Event, Event, BoolSort())
Next = Function('Next', Event, Event, BoolSort())
s.add(ForAll ([e1],(Next(e1, e1))))
s.add(ForAll ([e1,e2], Implies(And(Next(e1, e2),(e1 != e2)),Not(Next(e2, e1)))))
s.add(ForAll ([e1,e2,e3], Implies(And(And(Next(e1, e2),Next(e2, e3))),(Next(e1, e3)))))
s.add(ForAll ([e1,e2], Implies (And(iMNext(e1, e2),(e1 != e2)) ,Next(e1, e2))))
#--------------------------------
#Conflict Function
#--------------------------------
iConflict = Function('iConflict', Event, Event, BoolSort())
Conflict = Function('Conflict', Event, Event, BoolSort())
s.add(ForAll ([e1],(Not(Conflict(e1, e1)))))
s.add(ForAll ([e1,e2], Implies(And(Conflict(e1, e2),(e1 != e2)),Conflict(e2, e1))))
s.add(ForAll ([e1,e2,e3], Implies(And(And(Conflict(e1, e2),Next(e2, e3))),(Conflict(e1, e3))))
    )
s.add(ForAll ([e1,e2], Implies (And(iConflict(e1,e2),(e1 != e2)) ,Conflict(e1,e2))))
#--------------------------------
#Concurrency
#--------------------------------
Conc = Function('Conc', Event, Event, BoolSort())
s.add(ForAll([e1, e2],Conc(e1, e2) == Not(Or(Conflict(e1, e2),Next(e1, e2),Next(e2, e1)))))
#--------------------------------
#lifeline relation with events
#--------------------------------
cover = Function('cover', Lifeline, Event, BoolSort())
s.add(ForAll([L1, e1, L2], Implies(And (cover(L1, e1),(L1 != L2)), (Not(cover(L2, e1))))))
#--------------------------------
#Message  and its events relation
#--------------------------------
isMsg = Function ('isMsg', Event, Message, Event, BoolSort())
s.add(ForAll([e1,m,e2],Implies(isMsg(e1,m,e2),iMNext(e1,e2))))
s.add(ForAll([e1,m],(Not(isMsg(e1,m,e1)))))
#--------------------------------
#general order between events
```

```
#--------------------------------
s.add(iMNext(e1,e2))
s.add(iMNext(e2,e3))
s.add(iMNext(e3,e5))
s.add(iMNext(e5,e6))
s.add(iMNext(e6,e7))
s.add(iMNext(e7,e101))
s.add(iMNext(e3,e8))
s.add(iMNext(e8,e102))
s.add(iMNext(g1,g2))
s.add(iMNext(g2,g3))
s.add(iMNext(g3,g4))
s.add(iMNext(g4,g5))
s.add(iMNext(g5,g7))
s.add(iMNext(g5,g10))
s.add(iMNext(g7,g8))
s.add(iMNext(g8,g9))
s.add(iMNext(g9,g121))
s.add(iMNext(g10,g122))
s.add(iMNext(l1,l2))

s.add(iConflict(g7,g10))
s.add(iConflict(e5,e8))
#--------------------------------
# Connect message's to send event and receive event
#--------------------------------
s.add(isMsg(e1,InserCard,g1))
s.add(isMsg(g2,RequestPin,e2))
s.add(isMsg(e3,PinCode,g3))
s.add(isMsg(g4,Validate,l1))
s.add(isMsg(l2,Result,g5))
s.add(isMsg(g7,ValidPin,e5))
s.add(isMsg(g8,StartFuel,e6))
s.add(isMsg(e7,Stop,g9))
s.add(isMsg(g10,InvalidPin,e8))
s.add(isMsg(g121,CardOut,e101))
s.add(isMsg(g122,CardOut,e102))
#=========================
#assigning lifeline with its events
#=========================
s.add(cover(User,e1))
s.add(cover(User,e2))
s.add(cover(User,e3))
```

```
s.add(cover(User,e5))
s.add(cover(User,e6))
s.add(cover(User,e7))
s.add(cover(User,e8))
s.add(cover(User,e101))
s.add(cover(User,e102))
s.add(cover(PetrolStation,g1))
s.add(cover(PetrolStation,g2))
s.add(cover(PetrolStation,g3))
s.add(cover(PetrolStation,g4))
s.add(cover(PetrolStation,g5))
s.add(cover(PetrolStation,g7))
s.add(cover(PetrolStation,g8))
s.add(cover(PetrolStation,g9))
s.add(cover(PetrolStation,g10))
s.add(cover(PetrolStation,g121))
s.add(cover(PetrolStation,g122))
s.add(cover(Bank,l1))
s.add(cover(Bank,l2))
#---------------------------------
print (s.check())
print (s.model())
```

C.6 Z3 code for the woven model of the petrol station example in section 6.4

```
from z3 import *
#---------------------------------
#Lifeline declarations
#---------------------------------
Lifeline1 = DeclareSort('Lifeline1')
Lifeline2 = DeclareSort('Lifeline2')
Base_User = Const('Base_User', Lifeline1)
Base_PetrolStation = Const('Base_PetrolStation', Lifeline1)
Base_Bank = Const('Base_Bank', Lifeline1)
Advice_User = Const('Advice_User', Lifeline2)
Advice_PetrolStation = Const('Advice_PetrolStation', Lifeline2)
Advice_Bank = Const('Advice_Bank', Lifeline2)
```

```
#--------------------------------
#Declarations for the axioms
#--------------------------------
L_n = Const('L_n', Lifeline2)
L_k = Const('L_k', Lifeline2)
empty4 = Const('empty4', Lifeline2)
#--------------------------------
#Declarations for the axioms
#--------------------------------
L_i = Const('L_i', Lifeline1)
L_j = Const('L_j', Lifeline1)
empty3 = Const('empty3', Lifeline1)
#--------------------------------
#Events declarations
#--------------------------------
Event1 = DeclareSort('Event1')
Base_e1 = Const('Base_e1', Event1)
Base_e2 = Const('Base_e2', Event1)
Base_e3 = Const('Base_e3', Event1)
Base_e5 = Const('Base_e5', Event1)
Base_e6 = Const('Base_e6', Event1)
Base_e7 = Const('Base_e7', Event1)
Base_e8 = Const('Base_e8', Event1)
Base_e101 = Const('Base_e101', Event1)
Base_e102 = Const('Base_e102', Event1)
#================================
Base_g1 = Const('Base_g1', Event1)
Base_g2 = Const('Base_g2', Event1)
Base_g3 = Const('Base_g3', Event1)
Base_g4 = Const('Base_g4', Event1)
Base_g5 = Const('Base_g5', Event1)
Base_g7 = Const('Base_g7', Event1)
Base_g8 = Const('Base_g8', Event1)
Base_g9 = Const('Base_g9', Event1)
Base_g10 = Const('Base_g10', Event1)
Base_g121 = Const('Base_g121', Event1)
Base_g122 = Const('Base_g122', Event1)
#================================
Base_l1 = Const('Base_l1', Event1)
Base_l2 = Const('Base_l2', Event1)
#================================
#================================
Event2 = DeclareSort('Event2')
```

```
Advice_e1 = Const('Advice_e1', Event2)
Advice_e2 = Const('Advice_e2', Event2)
Advice_e3 = Const('Advice_e3', Event2)
Advice_e5 = Const('Advice_e5', Event2)
Advice_e6 = Const('Advice_e6', Event2)
#================================
Advice_g1 = Const('Advice_g1', Event2)
Advice_g2 = Const('Advice_g2', Event2)
Advice_g3 = Const('Advice_g3', Event2)
Advice_g4 = Const('Advice_g4', Event2)
Advice_g6 = Const('Advice_g6', Event2)
Advice_g7 = Const('Advice_g7', Event2)
Advice_g8 = Const('Advice_g8', Event2)
Advice_g9 = Const('Advice_g9', Event2)
Advice_g10 = Const('Advice_g10', Event2)
#================================
Advice_l1 = Const('Advice_l1', Event2)
Advice_l3 = Const('Advice_l3', Event2)
Advice_l4 = Const('Advice_l4', Event2)
Advice_l5 = Const('Advice_l5', Event2)
#================================
#Messages declarations
#================================
Message1 = DeclareSort('Message1')
Base_InserCard = Const('Base_InserCard', Message1)
Base_RequestPin = Const('Base_RequestPin', Message1)
Base_PinCode = Const('Base_PinCode', Message1)
Base_Validate = Const('Base_Validate', Message1)
Base_Result = Const('Base_Result', Message1)
Base_ValidPin = Const('Base_ValidPin', Message1)
Base_StartFuel = Const('Base_StartFuel', Message1)
Base_Stop = Const('Base_Stop', Message1)
Base_InvalidPin = Const('Base_InvalidPin', Message1)
Base_CardOut1 = Const('Base_CardOut1', Message1)
Base_CardOut2 = Const('Base_CardOut2', Message1)
#--------------------------------
#Declarations for the axioms
#--------------------------------
M_i = Const('M_i', Message1)
#M_j = Const('M_j', Message1)
empty5 = Const('empty5', Message1)
#--------------------------------
#--------------------------------
```

```
Message2 = DeclareSort('Message2')
Advice_Validate = Const('Advice_Validate', Message2)
Advice_EnterFuelAmount = Const('Advice_EnterFuelAmount', Message2)
Advice_FuelAmount = Const('Advice_FuelAmount', Message2)
Advice_CheckAmount = Const('Advice_CheckAmount', Message2)
Advice_BalanceOk = Const('Advice_BalanceOk', Message2)
Advice_StartFuel = Const('Advice_StartFuel', Message2)
Advice_Withdrew = Const('Advice_Withdrew', Message2)
Advice_Cancel = Const('Advice_Cancel', Message2)
Advice_PaymentDeclined = Const('Advice_PaymentDeclined', Message2)
Advice_ValidPin = Const('Advice_ValidPin', Message2)
#---------------------------------
#Declarations for the axioms
#---------------------------------
M_j = Const('M_j', Message2)
empty6 = Const('empty6', Message2)
#---------------------------------
#---------------------------------
#elements used for the constrants
#---------------------------------
L_i = Const('L_i', Lifeline1)
L_j = Const('L_j', Lifeline1)
L_n = Const('L_n', Lifeline2)
L_k = Const('L_k', Lifeline2)
empty1 = Const('empty1', Event1)
e_i = Const('e_i', Event1)
e_j = Const('e_j', Event1)
e_n = Const('e_n', Event1)
empty2 = Const('empty2', Event2)
g_i = Const('g_i', Event2)
g_j = Const('g_j', Event2)
g_n = Const('g_n', Event2)
M_i = Const('M_i', Message1)
M_j = Const('M_j', Message2)
#---------------------------------
list_e = [Base_e1,Base_e2,Base_e3,Base_e5,Base_e6,Base_e7,Base_e8,Base_e101,Base_e102,Base_g1,
    Base_g2,Base_g3,Base_g4,Base_g5,Base_g7,Base_g8,
Base_g9,Base_g10,Base_g121,Base_g122,Base_l1,Base_l2]
list_g = [Advice_e1,Advice_e2,Advice_e3,Advice_e5,Advice_e6,Advice_g1,Advice_g2,Advice_g3,
    Advice_g4,Advice_g6,Advice_g7,Advice_g8,Advice_g9,Advice_g10,
Advice_l1,Advice_l3,Advice_l4,Advice_l5]
ee1 = Distinct(Base_e1,Base_e2,Base_e3,Base_e5,Base_e6,Base_e7,Base_e8,Base_e101,Base_e102,
    Base_g1,Base_g2,Base_g3,Base_g4,Base_g5,Base_g7,Base_g8,
```

```
Base_g9,Base_g10,Base_g121,Base_g122,Base_l1,Base_l2)
ee2 = Distinct(Advice_e1,Advice_e2,Advice_e3,Advice_e5,Advice_e6,Advice_g1,Advice_g2,Advice_g3
    ,Advice_g4,Advice_g6,Advice_g7,Advice_g8,Advice_g9,Advice_g10,
Advice_l1,Advice_l3,Advice_l4,Advice_l5)
s.add(ee1,ee2,MM1,MM2)
def mklist(l1, l2):
result = []
for x in l1:
for y in l2:
result.append((x,y))
return result
def addPairs(s, relation, relation2,empty1, empty2, l1, l2, matches):
L = {}
R = {}
for (x, y) in matches:
L[str(x)] = True
R[str(y)] = True
for (x,y) in mklist(l1, l2):
if str((x,y)) in map(str, matches):
s.add(relation(x,y))
s.add(relation2(x,y))

else:
s.add(Not(relation(x,y)))
for x in l1:
if not str(x) in L:
s.add(relation(x, empty2))
s.add(Notmatch1(x))
else:
s.add(Not(Notmatch1(x)))
for x in l2:
if not str(x) in R:
s.add(relation(empty1, x))
s.add(Notmatch2(x))
else:
s.add(Not(Notmatch2(x)))
#print mklist(list_e, list_g)
s = Solver()
#================================================================
#MessagePresent and MessageMatch functions
#================================================================
MessagePresent = Function('MessagePresent', Message1,Message2, BoolSort())
MessageMatch = Function('MessageMatch ', Message1, Message2, BoolSort())
```

```
#=====================================================================
#EventMatch and present functions
#=====================================================================
EventMatch = Function('EventMatch ', Event1, Event2, BoolSort())
present = Function('present', Event1,Event2, BoolSort())
#=====================================================================
#LifelineMatch and LifelinePresent functions
#=====================================================================
LifelinePresent = Function('LifelinePresent', Lifeline1,Lifeline2, BoolSort())
LifelineMatch = Function('LifelineMatch', Lifeline1, Lifeline2, BoolSort())
#=====================================================================
#Notmatch  functions for Events of sd1 and sd2
#=====================================================================
Notmatch1 = Function('Notmatch1', Event1, BoolSort())
Notmatch2 = Function('Notmatch2', Event2, BoolSort())
#=====================================================================
#MessageNotmatch functions for Messages of sd1 and sd2
#=====================================================================
MessageNotmatch1 = Function('MessageNotmatch1', Message1, BoolSort())
MessageNotmatch2 = Function('MessageNotmatch2', Message2, BoolSort())
#=====================================================================
#NotMatchEventLifeline for Lifelines of sd1 and sd2
#=====================================================================
LifelineNotmatch1 = Function('LifelineNotmatch1', Lifeline1, BoolSort())
s.add(ForAll ([L_i,L_n], Implies(LifelineMatch(L_i,L_n),Not(LifelineNotmatch1(L_i)))))
LifelineNotmatch2 = Function('LifelineNotmatch2', Lifeline2, BoolSort())
s.add(ForAll ([L_i,L_n], Implies(LifelineMatch(L_i,L_n),Not(LifelineNotmatch2(L_n)))))
#---------------------------------
#Distinct
#---------------------------------
MM1 = Distinct(Base_InserCard,Base_RequestPin,Base_PinCode,Base_Validate,Base_Result,
    Base_ValidPin,Base_StartFuel,Base_Stop,Base_InvalidPin,Base_CardOut1,Base_CardOut2)
MM2 = Distinct(Advice_Validate,Advice_EnterFuelAmount,Advice_FuelAmount, Advice_CheckAmount,
    Advice_BalanceOk, Advice_StartFuel,Advice_Withdrew,Advice_Cancel,Advice_PaymentDeclined,
    Advice_ValidPin)
s.add(ee1,ee2,LL1,LL2,MM1,MM2)
#---------------------------------
#LES constraint,
#immediate Causality
#---------------------------------
iMNext1 = Function('iMNext1', Event1, Event1, BoolSort())
s.add(ForAll ([e_i],(Not(iMNext1(e_i, e_i)))))
#---------------------------------
```

```
#Causality ->*
#--------------------------------
Next1 = Function('Next1', Event1, Event1, BoolSort())
s.add(ForAll ([e_i],(Next1(e_i, e_i))))
s.add(ForAll ([e_i,e_j], Implies(And(Next1(e_i, e_j),(e_i != e_j)),Not(Next1(e_j, e_i)))))
s.add(ForAll ([e_i,e_j,e_n], Implies(And(Next1(e_i, e_j),Next1(e_j, e_n)),(Next1(e_i, e_n)))))
# All events connected by immediate Causality(iMNext) are connected by Causality relation (
    Next)
s.add(ForAll ([e_i,e_j], Implies (iMNext1(e_i,e_j) ,Next1(e_i,e_j))))
#=========================
#--------------------------------
#LES constraint,
#immediate Causality
#--------------------------------
iMNext2 = Function('iMNext2', Event2, Event2, BoolSort())
s.add(ForAll ([g_i],(Not(iMNext2(g_i, g_i)))))
#--------------------------------
#Causality ->*
#--------------------------------
Next2 = Function('Next2', Event2, Event2, BoolSort())
s.add(ForAll ([g_i],(Next2(g_i, g_i))))
s.add(ForAll ([g_i,g_j], Implies(And(Next2(g_i, g_j),(g_i != g_j)),Not(Next2(g_j, g_i)))))
s.add(ForAll ([g_i,g_j,g_n], Implies(And(Next2(g_i, g_j),Next2(g_j, g_n)),(Next2(g_i, g_n)))))
# All events connected by immediate Causality(iMNext) are connected by Causality relation (
    Next)
s.add(ForAll ([g_i,g_j], Implies (iMNext2(g_i,g_j) ,Next2(g_i,g_j))))
#=========================
#=========================
iMNext3  = Function('iMNext3 ', Event1,Event2,Event1, Event2, BoolSort())
s.add(ForAll ([e_i,g_i],Not(iMNext3(e_i,g_i,e_i,g_i))))
next3  = Function('next3 ', Event1,Event2,Event1, Event2, BoolSort())
s.add(ForAll ([e_i,g_i],(next3(e_i,g_i,e_i,g_i))))
s.add(ForAll ([e_i,g_i,e_j,g_j], Implies (And(next3(e_i,g_i,e_j,g_j),(e_i != e_j),(g_i != g_j)
    ),Not(next3(e_j,g_j,e_i,g_j)))))
s.add(ForAll ([e_i,g_i,e_j,g_j,e_n,g_n], Implies(And(next3(e_i,g_i,e_j,g_j),next3(e_j,g_j,e_n,
    g_n)),(next3(e_i,g_i,e_n,g_n)))))
s.add(ForAll ([e_i,g_i,e_j,g_j], Implies (iMNext3(e_i,g_i,e_j,g_j) ,next3(e_i,g_i,e_j,g_j))))
#=========================
#--------------------------------
# This just to show the direct conflict relation
#--------------------------------
iConflict1 = Function('iConflict1', Event1, Event1, BoolSort())
#--------------------------------
```

```
Conflict1 = Function('Conflict1', Event1, Event1, BoolSort())
s.add(ForAll ([e_i],(Not(Conflict1(e_i, e_i)))))
s.add(ForAll ([e_i,e_j], Implies(Conflict1(e_i, e_j),Conflict1(e_j, e_i))))
s.add(ForAll ([e_i,e_j,e_n], Implies(And(And(Conflict1(e_i, e_j),Next1(e_j, e_n))),(Conflict1(
    e_i, e_n)))))
# All events connected by immediate conflict(iConflict) are connected by conflict relation (
    Conflict)
s.add(ForAll ([e_i,e_j], Implies (iConflict1(e_i, e_j) ,Conflict1(e_i, e_j))))
#--------------------------------
# This just to show the direct conflict relation
#--------------------------------
iConflict2 = Function('iConflict2', Event2, Event2, BoolSort())
#--------------------------------
Conflict2 = Function('Conflict2', Event2, Event2, BoolSort())
s.add(ForAll ([g_i],(Not(Conflict2(g_i, g_i)))))
s.add(ForAll ([g_i,g_j], Implies(And(Conflict2(g_i, g_j),(g_i != g_j)),Conflict2(g_j, g_i))))
s.add(ForAll ([g_i,g_j,g_n], Implies(And(And(Conflict2(g_i, g_j),Next2(g_j, g_n))),(Conflict2(
    g_i, g_n)))))
# All events connected by immediate conflict(iConflict) are connected by conflict relation (
    Conflict)
s.add(ForAll ([g_i,g_j], Implies (iConflict2(g_i, g_j) ,Conflict2(g_i, g_j))))
#=========================
#=========================
iConflict3 = Function('iConflict3', Event1,Event2,Event1, Event2, BoolSort())
Conflict3 = Function('Conflict3', Event1,Event2,Event1, Event2, BoolSort())
s.add(ForAll ([e_i,g_i],(Not(Conflict3(e_i,g_i,e_i,g_i)))))
s.add(ForAll ([e_i,g_i,e_j,g_j], Implies (And(Conflict3(e_i,g_i,e_j,g_j),(e_i != e_j),(g_i !=
    g_j)),Conflict3(e_j,g_j,e_i,g_j))))
s.add(ForAll ([e_i,g_i,e_j,g_j,e_n,g_n], Implies(And(Conflict3(e_i,g_i,e_j,g_j),next3(e_j,g_j,
    e_n,g_n)),(Conflict3(e_i,g_i,e_n,g_n)))))
s.add(ForAll ([e_i,g_i,e_j,g_j], Implies (iConflict3(e_i,g_i,e_j,g_j) ,Conflict3(e_i,g_i,e_j,
    g_j))))
#--------------------------------
#messages constraint, the messages has only one event as receive and the event connect to only
     one messages
#--------------------------------
isMsg2 = Function('isMsg2',Event2, Message2, Event2, BoolSort())
s.add(ForAll([M_j, g_i],(Not(isMsg2(g_i,M_j,g_i)))))
s.add(ForAll([g_i,M_j,g_j],Implies(isMsg2(g_i,M_j,g_j),iMNext2(g_i,g_j))))
s.add(ForAll([g_i,M_j,g_j],Implies(And(And(isMsg2(g_i,M_j,g_j),iMNext2(g_i,g_j)),Notmatch2(g_i
    ),Notmatch2(g_j)),iMNext3(empty1,g_i,empty1,g_j))))
#--------------------------------
```

```
#messages constraint, the messages has only one event as receive and the event connect to only
     one messages
#--------------------------------
isMsg1 = Function('isMsg1',Event1, Message1, Event1, BoolSort())
s.add(ForAll([M_i, e_i],(Not(isMsg1(e_i,M_i,e_i)))))
s.add(ForAll([e_i,M_i,e_j],Implies(isMsg1(e_i,M_i,e_j),iMNext1(e_i,e_j))))
s.add(ForAll([e_i,M_i,e_j],Implies(And(And(isMsg1(e_i,M_i,e_j),iMNext1(e_i,e_j)),Notmatch1(e_i
    ),Notmatch1(e_j)),iMNext3(e_i,empty2,e_j,empty2))))
#--------------------------------
#Relation isMsg3 for composition
#--------------------------------
isMsg3 = Function('isMsg3',Event1,Event2, Message1,Message2, Event1,Event2, BoolSort())
#--------------------------------
#Matching axioms for messages
#--------------------------------
s.add(ForAll ([e_i,g_i,e_j,g_j,M_i,M_j], Implies
(And(And(MessageNotmatch1(M_i),MessageNotmatch2(M_j)),isMsg1(e_i,M_i,e_j),isMsg2(g_i,M_j,g_j))
    ,
And(isMsg3(e_i,empty2,M_i,empty6,e_j,empty2),isMsg3(empty1,g_i,empty5,M_j,empty1,g_j)))))
s.add(ForAll ([e_i,e_j,g_i,g_j,M_i,M_j], Implies (And(EventMatch(e_i,g_i),isMsg1(e_i,M_i,e_j),
    isMsg2(g_i,M_j,g_j)),MessageMatch (M_i, M_j))))
s.add(ForAll ([e_i,e_j,g_i,g_j,M_i,M_j], Implies (And(EventMatch(e_j,g_j),isMsg1(e_i,M_i,e_j),
    isMsg2(g_i,M_j,g_j)),MessageMatch (M_i, M_j))))
s.add(ForAll ([e_i,e_j,g_i,g_j,M_i,M_j], Implies (And(MessageMatch (M_i, M_j),isMsg1(e_i,M_i,
    e_j),isMsg2(g_i,M_j,g_j)),EventMatch(e_i,g_i))))
s.add(ForAll ([e_i,e_j,g_i,g_j,M_i,M_j], Implies (And(MessageMatch (M_i, M_j),isMsg1(e_i,M_i,
    e_j),isMsg2(g_i,M_j,g_j)),EventMatch (e_j, g_j))))
s.add(ForAll ([e_i,g_i,e_j,g_j,M_i,M_j], Implies
(And(And(MessageMatch (M_i, M_j)),isMsg1(e_i,M_i,e_j),isMsg2(g_i,M_j,g_j)),isMsg3(e_i,g_i,M_i,
    M_j,e_j,g_j))))
#--------------------------------
#cover relation for sd1
#--------------------------------
cover1 = Function('cover1', Lifeline1, Event1, BoolSort())
s.add(ForAll([L_i, e_i, L_j], Implies(And (cover1(L_i, e_i),(L_i != L_j)), (Not(cover1(L_j,
    e_i))))))
#--------------------------------
#cover relation for sd2
#--------------------------------
cover2 = Function('cover2', Lifeline2, Event2, BoolSort())
s.add(ForAll([L_n, g_i, L_k], Implies(And (cover2(L_n, g_i),(L_n != L_k)), (Not(cover2(L_k,
    g_i))))))
#========================
```

```
#cover relation for the composition
#========================
cover3 = Function('cover3', Lifeline1,Lifeline2, Event1,Event2, BoolSort())
#--------------------------------
#--------------------------------
#Matching axioms for lifelines
#--------------------------------
s.add(ForAll ([e_i,g_i,L_i,L_n], Implies (And(And(LifelineMatch(L_i, L_n),EventMatch(e_i,g_i))
    ,cover1(L_i,e_i),cover2(L_n,g_i)),cover3(L_i,L_n,e_i,g_i))))
s.add(ForAll ([e_i,g_i,L_i,L_n], Implies (And(And(And(cover1(L_i,e_i),cover2(L_n,g_i)),
    Notmatch1(e_i),Notmatch2(g_i)),LifelineNotmatch2(L_n),LifelineNotmatch1(L_i)),And(cover3(
    L_i,empty4,e_i,empty2),cover3(empty3,L_n,empty1,g_i)))))
s.add(ForAll ([e_i,L_i,L_n], Implies (And(LifelineMatch(L_i, L_n),cover1(L_i,e_i), Notmatch1(
    e_i)),cover3(L_i,L_n,e_i,empty2))))
s.add(ForAll ([g_i,L_i,L_n], Implies (And(LifelineMatch(L_i, L_n),cover2(L_n,g_i), Notmatch2(
    g_i)),cover3(L_i,L_n,empty1,g_i))))
s.add(ForAll ([e_i,g_i,L_i,L_n], Implies (And(And(EventMatch(e_i,g_i),cover1(L_i,e_i),cover2(
    L_n,g_i))),LifelineMatch(L_i,L_n))))
#--------------------------------
#--------------------------------
#Concurrency
#--------------------------------
Conc1 = Function('Conc1', Event1, Event1, BoolSort())
s.add(ForAll([e_i, e_j],Conc1(e_i, e_j)== Not(Or(Next1(e_i, e_j),Next1(e_j, e_i),Conflict1(e_i
    , e_j)))))
Conc2 = Function('Conc2', Event2, Event2, BoolSort())
s.add(ForAll([g_i, g_j], Conc2(g_i, g_j)== Not(Or(Next2(g_i, g_j),Next2(g_j, g_i),Conflict2(
    g_i, g_j)))))
Conc3 = Function('Conc3', Event1,Event2,Event1, Event2, BoolSort())
s.add(ForAll([e_i,g_i,e_j,g_j],Conc3(e_i,g_i,e_j,g_j)== Not(Or(next3(e_i,g_i,e_j,g_j),next3(
    e_j,g_j,e_i,g_j),Conflict3(e_i,g_i,e_j,g_j)))))
#-------------------------------------------------------------
#Matching axioms for Events in Causality relation
#-------------------------------------------------------------
s.add(ForAll ([e_i,g_i,e_j,g_j], Implies(And(And(iMNext1(e_i, e_j),iMNext2(g_i,g_j)),
    EventMatch(e_i, g_i),EventMatch(e_j, g_j)),iMNext3(e_i,g_i,e_j,g_j))))
s.add(ForAll ([e_i,g_i,e_j,g_j], Implies (And(And(And(iMNext1(e_i, e_j),iMNext2(g_i,g_j)),
    Notmatch1(e_i),Notmatch2(g_i)),Notmatch1(e_j),Notmatch2(g_j)),And(iMNext3(e_i,empty2,e_j,
    empty2),iMNext3(empty1,g_i,empty1,g_j)))))
s.add(ForAll ([e_i,g_i,e_j], Implies(And(And(EventMatch(e_i,g_i),iMNext1(e_i,e_j)),Notmatch1(
    e_j)),iMNext3(e_i,g_i,e_j,empty2))))
s.add(ForAll ([e_i,g_i,g_j], Implies(And(And(EventMatch(e_i,g_i),iMNext2(g_i,g_j)),Notmatch2(
    g_j)),iMNext3(e_i,g_i,empty1,g_j))))
```

```
s.add(ForAll ([e_i,g_j,e_j], Implies(And(And(EventMatch(e_j, g_j),iMNext1(e_i,e_j)),Notmatch1(
    e_i)),iMNext3(e_i,empty2,e_j,g_j))))
s.add(ForAll ([e_j,g_i,g_j], Implies(And(And(EventMatch(e_j, g_j),iMNext2(g_i,g_j)),Notmatch2(
    g_i)),iMNext3(empty1,g_i,e_j,g_j))))
#-------------------------------------------------------------
#Matching axioms for Events in iConflict1 relation
#-------------------------------------------------------------
s.add(ForAll ([e_i,g_i,e_j,g_j], Implies(And(And(iConflict1(e_i, e_j),iConflict2(g_i,g_j)),
    EventMatch(e_i, g_i),EventMatch(e_j, g_j)),iConflict3(e_i,g_i,e_j,g_j))))
s.add(ForAll ([e_i,g_i,e_j,g_j], Implies (And(And(And(iConflict1(e_i, e_j),iConflict2(g_i,g_j)
    ),Notmatch1(e_i),Notmatch2(g_i)),Notmatch1(e_j),Notmatch2(g_j)),And(iConflict3(e_i,empty2,
    e_j,empty2),iConflict3(empty1,g_i,empty1,g_j)))))
s.add(ForAll ([e_i,g_i,e_j], Implies(And(And(EventMatch(e_i,g_i),iConflict1(e_i,e_j)),
    Notmatch1(e_j)),iConflict3(e_i,g_i,e_j,empty2))))
s.add(ForAll ([e_i,g_i,g_j], Implies(And(And(EventMatch(e_i,g_i),iConflict2(g_i,g_j)),
    Notmatch2(g_j)),iConflict3(e_i,g_i,empty1,g_j))))
s.add(ForAll ([e_i,g_j,e_j], Implies(And(And(EventMatch(e_j, g_j),iConflict1(e_i,e_j)),
    Notmatch1(e_i)),iConflict3(e_i,empty2,e_j,g_j))))
s.add(ForAll ([e_j,g_i,g_j], Implies(And(And(EventMatch(e_j, g_j),iConflict2(g_i,g_j)),
    Notmatch2(g_i)),iConflict3(empty1,g_i,e_j,g_j))))


#---------------------------------
#general order between events
#---------------------------------
s.add(iMNext2(Advice_e1,Advice_e2))
s.add(iMNext2(Advice_e2,Advice_e3))
s.add(iMNext2(Advice_e3,Advice_e5))
s.add(iMNext2(Advice_e3,Advice_e6))
#=========================
s.add(iMNext2(Advice_g1,Advice_g2))
s.add(iMNext2(Advice_g2,Advice_g3))
s.add(iMNext2(Advice_g3,Advice_g4))
s.add(iMNext2(Advice_g4,Advice_g6))
s.add(iMNext2(Advice_g6,Advice_g7))
s.add(iMNext2(Advice_g7,Advice_g8))
s.add(iMNext2(Advice_g4,Advice_g9))
s.add(iMNext2(Advice_g9,Advice_g10))
#=========================
s.add(iMNext2(Advice_l1,Advice_l3))
s.add(iMNext2(Advice_l3,Advice_l4))
s.add(iMNext2(Advice_l1,Advice_l5))
s.add(iConflict2(Advice_e5,Advice_e6))
s.add(iConflict2(Advice_g6,Advice_g9))
```

```
s.add(iConflict2(Advice_l3,Advice_l5))
#=========================
#=========================
s.add(iMNext1(Base_e1,Base_e2))
s.add(iMNext1(Base_e2,Base_e3))
s.add(iMNext1(Base_e3,Base_e5))
s.add(iMNext1(Base_e5,Base_e6))
s.add(iMNext1(Base_e6,Base_e7))
s.add(iMNext1(Base_e7,Base_e101))
s.add(iMNext1(Base_e3,Base_e8))
s.add(iMNext1(Base_e8,Base_e102))
s.add(iMNext1(Base_g1,Base_g2))
s.add(iMNext1(Base_g2,Base_g3))
s.add(iMNext1(Base_g3,Base_g4))
s.add(iMNext1(Base_g4,Base_g5))
s.add(iMNext1(Base_g5,Base_g7))
s.add(iMNext1(Base_g5,Base_g10))
s.add(iMNext1(Base_g7,Base_g8))
s.add(iMNext1(Base_g8,Base_g9))
s.add(iMNext1(Base_g9,Base_g121))
s.add(iMNext1(Base_g10,Base_g122))
s.add(iMNext1(Base_l1,Base_l2))
s.add(iConflict1(Base_e5,Base_e8))
s.add(iConflict1(Base_g7,Base_g10))
#=========================
# assigning messages with its events
#=========================
s.add(isMsg2(Advice_g1,Advice_ValidPin,Advice_e1))
s.add(isMsg2(Advice_g2,Advice_EnterFuelAmount,Advice_e2))
s.add(isMsg2(Advice_e3,Advice_FuelAmount,Advice_g3))
s.add(isMsg2(Advice_g4,Advice_CheckAmount,Advice_l1))
s.add(isMsg2(Advice_g8,Advice_StartFuel,Advice_e5))
s.add(isMsg2(Advice_g10,Advice_PaymentDeclined,Advice_e6))
s.add(isMsg2(Advice_l4,Advice_Withdrew,Advice_g7))
s.add(isMsg2(Advice_l3,Advice_BalanceOk,Advice_g6))
s.add(isMsg2(Advice_l5,Advice_Cancel,Advice_g9))
#=========================
s.add(isMsg1(Base_e1,Base_InserCard,Base_g1))
s.add(isMsg1(Base_g2,Base_RequestPin,Base_e2))
s.add(isMsg1(Base_e3,Base_PinCode,Base_g3))
s.add(isMsg1(Base_g4,Base_Validate,Base_l1))
s.add(isMsg1(Base_l2,Base_Result,Base_g5))
s.add(isMsg1(Base_g7,Base_ValidPin,Base_e5))
```

```
s.add(isMsg1(Base_g8,Base_StartFuel,Base_e6))
s.add(isMsg1(Base_e7,Base_Stop,Base_g9))
s.add(isMsg1(Base_g10,Base_InvalidPin,Base_e8))
s.add(isMsg1(Base_g121,Base_CardOut1,Base_e101))
s.add(isMsg1(Base_g122,Base_CardOut2,Base_e102))
#=========================
#assigning lifeline with its events
#=========================
s.add(cover2(Advice_User,Advice_e1))
s.add(cover2(Advice_User,Advice_e2))
s.add(cover2(Advice_User,Advice_e3))
s.add(cover2(Advice_User,Advice_e5))
s.add(cover2(Advice_User,Advice_e6))
s.add(cover2(Advice_PetrolStation,Advice_g1))
s.add(cover2(Advice_PetrolStation,Advice_g2))
s.add(cover2(Advice_PetrolStation,Advice_g3))
s.add(cover2(Advice_PetrolStation,Advice_g4))
s.add(cover2(Advice_PetrolStation,Advice_g6))
s.add(cover2(Advice_PetrolStation,Advice_g7))
s.add(cover2(Advice_PetrolStation,Advice_g8))
s.add(cover2(Advice_PetrolStation,Advice_g9))
s.add(cover2(Advice_PetrolStation,Advice_g10))
s.add(cover2(Advice_Bank,Advice_l1))
s.add(cover2(Advice_Bank,Advice_l3))
s.add(cover2(Advice_Bank,Advice_l4))
s.add(cover2(Advice_Bank,Advice_l5))
#-------------------------------
s.add(cover1(Base_User,Base_e1))
s.add(cover1(Base_User,Base_e2))
s.add(cover1(Base_User,Base_e3))
s.add(cover1(Base_User,Base_e5))
s.add(cover1(Base_User,Base_e6))
s.add(cover1(Base_User,Base_e7))
s.add(cover1(Base_User,Base_e8))
s.add(cover1(Base_User,Base_e101))
s.add(cover1(Base_User,Base_e102))
s.add(cover1(Base_PetrolStation,Base_g1))
s.add(cover1(Base_PetrolStation,Base_g2))
s.add(cover1(Base_PetrolStation,Base_g3))
s.add(cover1(Base_PetrolStation,Base_g4))
s.add(cover1(Base_PetrolStation,Base_g5))
s.add(cover1(Base_PetrolStation,Base_g7))
s.add(cover1(Base_PetrolStation,Base_g8))
```

```
s.add(cover1(Base_PetrolStation,Base_g9))
s.add(cover1(Base_PetrolStation,Base_g10))
s.add(cover1(Base_PetrolStation,Base_g121))
s.add(cover1(Base_PetrolStation,Base_g122))
s.add(cover1(Base_Bank,Base_l1))
s.add(cover1(Base_Bank,Base_l2))
cover_info_event_message = {}
messageMatches = {}
allMessage1 = {}
allMessage2 = {}
#--------------------------------
# Process Message matches
# ==== START
assertions = s.assertions()
for ast in assertions:
if not "is_forall" in dir(ast) and str(ast.decl()) in ["isMsg1", "isMsg2"]:
cover_info_event_message[str(ast.arg(0))] = ast.arg(1)
cover_info_event_message[str(ast.arg(2))] = ast.arg(1)
if "1" in str(ast.arg(1).sort()):
allMessage1[str(ast.arg(1))] = ast.arg(1)
if "2" in str(ast.arg(1).sort()):
allMessage2[str(ast.arg(1))] = ast.arg(1)
for (x,y) in matches:
oldx = x
oldy = y
x1 = cover_info_event_message[str(oldx)]
y1 = cover_info_event_message[str(oldy)]

pair11 = (str(x1), str(y1))
pair12 = (str(y1), str(x1))
messageMatches[str(pair11)] = True
messageMatches[str(pair12)] = True
allMessagePairs = mklist(allMessage1, allMessage2)
messageMatched = {}
for (x, y) in allMessagePairs:
x = allMessage1[x]
y = allMessage2[y]
if str(x) == str(y): continue
if str((str(x), str(y))) in messageMatches:
s.add(MessageMatch(x, y))
s.add(MessagePresent(x, y))

messageMatched[str(x)]=True
```

```
messageMatched[str(y)]=True
else:
s.add(Not(MessagePresent(x, y)))

for l in allMessage1:
if not str(l) in messageMatched:
s.add(MessagePresent(allMessage1[l], empty6))
s.add(MessageNotmatch1(allMessage1[l]))
else:
s.add(Not(MessageNotmatch1(allMessage1[l])))
for l in allMessage2:
if not str(l) in messageMatched:
s.add(MessagePresent(empty5, allMessage2[l]))
s.add(MessageNotmatch2(allMessage2[l]))
else:
s.add(Not(MessageNotmatch2(allMessage2[l])))
# ==== END
cover_info_event_lineline = {}
lifelineMatches = {}
allLifelines1 = {}
allLifelines2 = {}
#--------------------------------
# Process lifeline matches
# ==== START
assertions = s.assertions()
for ast in assertions:
if not "is_forall" in dir(ast) and str(ast.decl()) in ["cover1", "cover2"]:
cover_info_event_lineline[str(ast.arg(1))] = ast.arg(0)
if "1" in str(ast.arg(0).sort()):
allLifelines1[str(ast.arg(0))] = ast.arg(0)
if "2" in str(ast.arg(0).sort()):
allLifelines2[str(ast.arg(0))] = ast.arg(0)
for (x,y) in matches:
oldx = x
oldy = y
x = cover_info_event_lineline[str(x)]
y = cover_info_event_lineline[str(y)]
pair1 = (str(x), str(y))
pair2 = (str(y), str(x))
lifelineMatches[str(pair1)] = True
lifelineMatches[str(pair2)] = True
allLifelinePairs = mklist(allLifelines1, allLifelines2)
lifelineMatched = {}
```

```
for (x, y) in allLifelinePairs:
x = allLifelines1[x]
y = allLifelines2[y]
if str(x) == str(y): continue
if str((str(x), str(y))) in lifelineMatches:
s.add(LifelineMatch(x, y))
s.add(LifelinePresent(x, y))
lifelineMatched[str(x)]=True
lifelineMatched[str(y)]=True
else:
s.add(Not(LifelineMatch(x, y)))
s.add(Not(LifelinePresent(x, y)))

for l in allLifelines1:
if not str(l) in lifelineMatched:
s.add(LifelineNotmatch1(allLifelines1[l]))
s.add(LifelinePresent(allLifelines1[l], empty4))

else:
s.add(Not(LifelineNotmatch1(allLifelines1[l])))
s.add(Not(LifelinePresent(allLifelines1[l], empty4)))

for l in allLifelines2:
if not str(l) in lifelineMatched:
s.add(LifelineNotmatch2(allLifelines2[l]))
s.add(LifelinePresent(empty3, allLifelines2[l]))

else:
s.add(Not(LifelineNotmatch2(allLifelines2[l])))
s.add(Not(LifelinePresent(empty3, allLifelines2[l])))
# ==== END
addPairs(s, present, EventMatch, empty1, empty2, list_e, list_g, matches)
#--------------------------------
#Pointcut
#=========================
# Message Match (Base_ValidPin, Advice_ValidPin))
# Message Match (Base_StartFuel, Advice_StartFuel))
s.add (LifelineMatch (Base_Bank, Advice_Bank))
matches = [(Base_e5,Advice_e1),(Base_e6,Advice_e5),(Base_g7,Advice_g1),(Base_g8,Advice_g8)]
print (s.check())
```

LIST OF REFERENCES

[1] Poseidon for UML [online]. Available at: `http://www.gentleware.com/`. [Accessed 04 May 2013].

[2] The Xactium XMF mosaic [online]. Available at: `http://www.xactium.com`. [Accessed 01 December 2015].

[3] David H. Akehurst, Behzad Bordbar, Michael J. Evans, Gareth J. Howells, and Klaus D. McDonald-Maier. Sitra: Simple transformations in java. In *Model Driven Engineering Languages and Systems, 9th International Conference, MoDELS 2006, Genova, Italy, October 1-6, 2006, Proceedings*, pages 351–364, 2006.

[4] Mohammed Alodib, Behzad Bordbar, and Basim Majeed. A model driven approach to the design and implementing of fault tolerant service oriented architectures. In *Third IEEE International Conference on Digital Information Management (ICDIM), November 13-16, 2008, London, UK, Proceedings*, pages 464–469, 2008.

[5] Mohammed Alwanain, Behzad Bordbar, and Juliana Küster Filipe Bowles. Automated composition of sequence diagrams via alloy. In *MODELSWARD 2014 - Proceedings of the 2nd International Conference on Model-Driven Engineering and Software Development, Lisbon, Portugal, 7 - 9 January, 2014*, pages 384–391, 2014.

[6] Mohamed A. Ameedeen. *A model driven approach to analysis and synthesis of sequence diagrams*. PhD thesis, University of Birmingham, 2012.

[7] Mohamed A. Ameedeen, Behzad Bordbar, and Rachid Anane. Model interoperability via model driven development. *J. Comput. Syst. Sci.*, 77(2):332–347, 2011.

[8] Kyriakos Anastasakis. *A model driven approach for the automated analysis of UML class diagrams*. PhD thesis, University of Birmingham, 2009.

[9] Kyriakos Anastasakis, Behzad Bordbar, Geri Georg, and Indrakshi Ray. Uml2alloy: A challenging model transformation. In *Model Driven Engineering Languages and Systems, 10th International Conference, MoDELS 2007, Nashville, USA, September 30 - October 5, 2007, Proceedings*, pages 436–450, 2007.

[10] João Araújo and Jon Whittle. Aspect-oriented compositions for dynamic behavior models. In *Aspect-Oriented Requirements Engineering*, pages 45–60. 2013.

[11] João Araújo, Jon Whittle, and Dae-Kyoo Kim. Modeling and composing scenario-based requirements with aspects. In *12th IEEE International Conference on Requirements Engineering (RE 2004), 6-10 September 2004, Kyoto, Japan*, pages 58–67, 2004.

[12] Alessandro Armando, Jacopo Mantovani, and Lorenzo Platania. Bounded model checking of software using SMT solvers instead of SAT solvers. *STTT*, 11(1):69–83, 2009.

[13] Colin Atkinson and Ralph Gerbig. Aspect-oriented concrete syntax definition for deep modeling languages. In *Proceedings of the 2nd International Workshop on Multi-Level Modelling co-located with ACM/IEEE 18th International Conference on Model Driven Engineering Languages & Systems (MoDELS 2015), Ottawa, Canada, September 27-28, 2015.*, pages 13–22, 2015.

[14] Martin Auer, T. Tschurtschenthaler, and Stefan Biffl. A flyweight UML modelling tool for software development in heterogeneous environments. In *29th EUROMICRO Conference 2003, New Waves in System Architecture, 3-5 September 2003, Belek-Antalya, Turkey*, pages 267–272, 2003.

[15] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008.

[16] Clark Barrett, Aaron Stump, and Cesare Tinelli. The satisfiability modulo theories library (smt-lib). www. *SMT-LIB. org*, 15:18–52, 2010.

[17] Daniel Le Berre and Anne Parrain. The Sat4j library, release 2.2. *Journal on Satisfiability, Boolean Modeling and Computation*, 7(2-3):59–6, 2010.

[18] Armin Biere. Picosat essentials. *Journal on Satisfiability, Boolean Modeling and Computation*, 4:75–97, 2008.

[19] Thomas Bouton, Diego Caminha B. de Oliveira, David Déharbe, and Pascal Fontaine. verit: An open, trustable and efficient smt-solver. In *Automated Deduction - CADE-22, 22nd International Conference on Automated Deduction, Montreal, Canada, August 2-7, 2009. Proceedings*, pages 151–156, 2009.

[20] Juliana Bowles. Decomposing interactions. In *Algebraic Methodology and Software Technology, 11th International Conference, AMAST 2006, Kuressaare, Estonia, July 5-8, 2006, Proceedings*, pages 189–203, 2006.

[21] Juliana Bowles, Mohammed Alwanain, Behzad Bordbar, and Y. Chen. Matching and merging scenarios automatically with alloy. In *Model-Driven Engineering and Software Development - Second International Conference, MODELSWARD 2014, Lisbon, Portugal, January 7-9, 2014, Revised Selected Papers*, pages 100–116, 2014.

[22] Juliana Bowles and Behzad Bordbar. A formal model for integrating multiple views. In *Seventh International Conference on Application of Concurrency to System Design (ACSD 2007), 10-13 July 2007, Bratislava, Slovak Republic*, pages 71–79, 2007.

[23] Juliana Bowles, Behzad Bordbar, and Mohammed Alwanain. A logical approach for behavioural composition of scenario-based models. In *Formal Methods and Software Engineering - 17th International Conference on Formal Engineering Methods, ICFEM 2015, Paris, France, November 3-5, 2015, Proceedings*, pages 252–269, 2015.

[24] Alessandra Cavarra and Juliana Bowles. Formalizing liveness-enriched sequence diagrams using asms. In *Abstract State Machines 2004. Advances in Theory and Practice, 11th International Workshop, ASM 2004, Lutherstadt Wittenberg, Germany, May 24-28, 2004. Proceedings*, pages 62–77, 2004.

[25] Alessandra Cavarra and Juliana Bowles. Combining sequence diagrams and OCL for liveness. *Theoretical Computer Science*, 115:19–38, 2005.

[26] María Victoria Cengarle, Peter Graubmann, and Stefan Wagner. Semantics of UML 2.0 interactions with variabilities. *Theoretical Computer Science*, 160:141–155, 2006.

[27] Marıa Victoria Cengarle and Alexander Knapp. Uml 2.0 interactions: Semantics and refinement. In *Proc. 3rd Int. Wsh. Critical Systems Development with UML (CSDUML04)*, pages 85–99. Citeseer, 2004.

[28] Daniel Yuh Chao. Knitting technique with TP-PT generations for petri net synthesis. *Journal of information science and engineering*, 22(4):909–923, 2006.

[29] Marsha Chechik, Shiva Nejati, and Mehrdad Sabetzadeh. A relationship-based approach to model integration. *ISSE*, 8(1):3–18, 2012.

[30] Michele Chinosi and Alberto Trombetta. Bpmn: An introduction to the standard. *Computer Standards & Interfaces*, 34(1):124–134, 2012.

[31] Tony Clark and Pierre-Alain Muller. Exploiting model driven technology: a tale of two startups. *Software and System Modeling*, 11(4):481–493, 2012.

[32] Tony Clark, Paul Sammut, and James S. Willans. Applied metamodelling: A foundation for language driven development (third edition). *CoRR*, abs/1505.00149, 2015.

[33] Siobhán Clarke. *Composition of object-oriented software design models*. PhD thesis, Dublin City University, 2001.

[34] Mickael Clavreul. *Model and Metamodel Composition: Separation of Mapping and Interpretation for Unifying Existing Model Composition Techniques*. PhD thesis, Université Rennes, 2011.

[35] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, May 3-5, 1971, Shaker Heights, Ohio, USA*, pages 151–158, 1971.

[36] Lucas C. Cordeiro and Bernd Fischer. Verifying multi-threaded software using smt-based context-bounded model checking. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, pages 331–340, 2011.

[37] Luigi Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. An improved algorithm for matching large graphs. In *3rd IAPR-TC15 workshop on graph-based representations in pattern recognition*, pages 149–159, 2001.

[38] Luigi P Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. A (sub) graph isomorphism algorithm for matching large graphs. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 26(10):1367–1372, 2004.

[39] Thomas Cottenier, Aswin Van Den Berg, and Tzilla Elrad. The motorola weavr: Model weaving in a large industrial context. *Aspect-Oriented Software Development (AOSD), Vancouver, Canada*, 32:44, 2007.

[40] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM (JACM)*, 7(3):201–215, 1960.

[41] Leonardo Mendonça de Moura and Nikolaj Bjørner. Efficient e-matching for SMT solvers. In *Automated Deduction - CADE-21, 21st International Conference on Automated Deduction, Bremen, Germany, July 17-20, 2007, Proceedings*, pages 183–198, 2007.

[42] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, pages 337–340, 2008.

[43] Mu Der Jeng and Frank DiCesare. A review of synthesis techniques for petri nets with

applications to automated manufacturing systems. *Systems, Man and Cybernetics, IEEE Transactions on*, 23(1):301–312, 1993.

[44] Morgan Deters, Andrew Reynolds, Tim King, Clark W. Barrett, and Cesare Tinelli. A tour of CVC4: how it works, and how to use it. In *Formal Methods in Computer-Aided Design, FMCAD 2014, Lausanne, Switzerland, October 21-24, 2014*, page 7, 2014.

[45] Zhi-Jun Ding, Jun-Li Wang, and Chang-Jun Jiang. An approach for synthesis petri nets for modeling and verifying composite web service. *Journal of Information Science and Engineering*, 24(5):1309–1328, 2008.

[46] Nicolás D'Ippolito, Marcelo F. Frias, Juan P. Galeotti, Esteban Lanzarotti, and Sergio Mera. Alloy+hotcore: A fast approximation to unsat core. In *Abstract State Machines, Alloy, B and Z, Second International Conference, ABZ 2010, Orford, QC, Canada, February 22-25, 2010. Proceedings*, pages 160–173, 2010.

[47] Bruno Dutertre and Leonardo De Moura. The yices smt solver. *Tool paper at http://yices. csl. sri. com/tool-paper. pdf*, 2(2), 2006.

[48] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In *Theory and applications of satisfiability testing*, pages 502–518. Springer, 2003.

[49] Christoph Eichner, Hans Fleischhack, Roland Meyer, Ulrik Schrimpf, and Christian Stehno. Compositional semantics for uml 2.0 sequence diagrams using petri nets. In *SDL 2005: Model Driven*, pages 133–148. Springer, 2005.

[50] John Ellson, Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Gordon Woodhull. Graphviz - open source graph drawing tools. In *Graph Drawing*, pages 483–484, 2001.

[51] Javier Esparza. Reduction and synthesis of live and bounded free choice petri nets. *Inf. Comput.*, 114(1):50–87, 1994.

[52] Jean-Rémy Falleri, Marianne Huchard, and Clémentine Nebut. Towards a traceability framework for model transformations in kermeta. In *ECMDA-TW'06: ECMDA Traceability Workshop*, pages 31–40. Sintef ICT, Norway, 2006.

[53] Joao M. Fernandes, Simon Tjell, Jens Bæk Jørgensen, and Oscar Ribeiro. Designing tool support for translating use cases and uml 2.0 sequence diagrams into a coloured petri net. In *Scenarios and State Machines, 2007. SCESM'07: ICSE Workshops 2007. Sixth International Workshop on*, pages 2–2. IEEE, 2007.

[54] Juliana Küster Filipe. Modelling concurrent interactions. *Theoretical Computer Science*, 351(2):203–220, 2006.

[55] Franck Fleurey. Kompose : a generic model composition tool. Available at: `http://www.kermeta.org/kompose/`. [Accessed 22 September 2013].

[56] Franck Fleurey, Benoit Baudry, Robert B. France, and Sudipto Ghosh. A generic approach for automatic model composition. In *Models in Software Engineering, Workshops and Symposia at MoDELS 2007, Nashville, TN, USA, September 30 - October 5, 2007, Reports and Revised Selected Papers*, pages 7–15, 2007.

[57] Robert B. France, Indrakshi Ray, Geri Georg, and Sudipto Ghosh. Aspect-oriented approach to early design modelling. *IEE Proceedings - Software*, 151(4):173–186, 2004.

[58] Ana Gabriela Garis, Alcino Cunha, and Daniel Riesco. Translating alloy specifications to UML class diagrams annotated with OCL. In *Software Engineering and Formal Methods - 9th International Conference, SEFM 2011, Montevideo, Uruguay, November 14-18, 2011. Proceedings*, pages 221–236, 2011.

[59] Ana Gabriela Garis, Ana C. R. Paiva, Alcino Cunha, and Daniel Riesco. Specifying UML protocol state machines in alloy. In *Integrated Formal Methods - 9th International Conference, IFM 2012, Pisa, Italy, June 18-21, 2012. Proceedings*, pages 312–326, 2012.

[60] Emsaieb Geepalla, Behzad Bordbar, and Joel Last. Transformation of spatio-temporal role based access control specification to alloy. In *Model and Data Engineering - 2nd International Conference, MEDI 2012, Poitiers, France, October 3-5, 2012. Proceedings*, pages 67–78, 2012.

[61] Emsaieb Mosbah Geepalla. *Model-driven approaches to analysing time-and location-dependent access control specifications*. PhD thesis, University of Birmingham, 2013.

[62] Christian Gerth, Jochen Malte Küster, Markus Luckey, and Gregor Engels. Detection and resolution of conflicting change operations in version management of process models. *Software and System Modeling*, 12(3):517–535, 2013.

[63] Aboubakr Achraf El Ghazi and Mana Taghdiri. Relational reasoning via SMT solving. In *FM 2011: Formal Methods - 17th International Symposium on Formal Methods, Limerick, Ireland, June 20-24, 2011. Proceedings*, pages 133–148, 2011.

[64] Eugene Goldberg and Yakov Novikov. Berkmin: A fast and robust sat-solver. *Discrete Applied Mathematics*, 155(12):1549–1561, 2007.

[65] Cesar Gonzalez-Perez and Brian Henderson-Sellers. *Metamodelling for software engineering*. Wiley, 2008.

[66] Florian Gottschalk, Wil M. P. van der Aalst, and Monique H. Jansen-Vullers. Merging event-driven process chains. In *On the Move to Meaningful Internet Systems: OTM 2008, OTM 2008 Confederated International Conferences, CoopIS, DOA, GADA, IS, and ODBASE 2008, Monterrey, Mexico, November 9-14, 2008, Proceedings, Part I*, pages 418–426, 2008.

[67] Iris Groher and Markus Voelter. Xweave: models and aspects in concert. In *Proceedings of the 10th international workshop on Aspect-oriented modeling*, pages 35–40. ACM, 2007.

[68] Roy Grønmo and Birger Møller-Pedersen. From UML 2 sequence diagrams to state machines by graph transformation. *Journal of Object Technology*, 10:8: 1–22, 2011.

[69] Roy Grønmo, Ragnhild Kobro Runde, and Birger Møller-Pedersen. Confluence of aspects for sequence diagrams. *Software and System Modeling*, 12(4):789–824, 2013.

[70] Roy Grønmo, Fredrik Sørensen, Birger Møller-Pedersen, and Stein Krogdahl. Semantics-based weaving of UML sequence diagrams. In *Theory and Practice of Model Transformations, First International Conference, ICMT 2008, Zürich, Switzerland, July 1-2, 2008, Proceedings*, pages 122–136, 2008.

[71] Serge Haddad. A reduction theory for coloured nets. In *Advances in Petri Nets 1989, covers the 9th European Workshop on Applications and Theory in Petri Nets, held in Venice, Italy in June 1988, selected papers*, pages 209–235, 1988.

[72] Rachid Hamadi and Boualem Benatallah. A petri net-based model for web service composition. In *Database Technologies 2003, Proceedings of the 14th Australasian Database Conference, ADC 2003, Adelaide, South Australia, February 2003*, pages 191–200, 2003.

[73] Mounira Kezadri Hamiaz, Marc Pantel, Xavier Thirioux, and Benoît Combemale. Correct-by-construction model driven engineering composition operators. *Formal Asp. Comput.*, 28(3):409–440, 2016.

[74] Youcef Hammal. Branching time semantics for UML 2.0 sequence diagrams. In *Formal Techniques for Networked and Distributed Systems - FORTE 2006, 26th IFIP WG 6.1 International Conference, Paris, France, September 26-29, 2006.*, pages 259–274, 2006.

[75] Wu Hao. *Automated Metamodel Instance Generation Satisfying Quantitative Constraints*. PhD thesis, National University of Ireland Maynooth, 2013.

[76] David Harel. Statecharts: A visual formalism for complex systems. *Science of computer programming*, 8(3):231–274, 1987.

[77] David Harel. *Come, let's play - scenario-based programming using LSCs and the play-engine*. Springer, 2003.

[78] David Harel and Shahar Maoz. Assert and negate revisited: Modal semantics for UML sequence diagrams. *Software and System Modeling*, 7(2):237–252, 2008.

[79] Øystein Haugen, Knut Eilif Husa, Ragnhild Kobro Runde, and Ketil Stølen. Why timed sequence diagrams require three-event semantics. In *Scenarios: Models, Transformations and Tools, International Workshop, Dagstuhl Castle, Germany, September 7-12, 2003, Revised Selected Papers*, pages 1–25, 2003.

[80] Øystein Haugen, Knut Eilif Husa, Ragnhild Kobro Runde, and Ketil Stølen. STAIRS towards formal design with sequence diagrams. *Software and System Modeling*, 4(4):355–367, 2005.

[81] Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):256–290, 2002.

[82] Daniel Jackson. *Software Abstractions: logic, language and analysis*. MIT Press, 2006.

[83] Eyoun Jacobsen. *Concepts and Language Mechanisms in Software Modelling*. PhD thesis, University of Southern Denmark., 2000.

[84] Amin Jalali. Static weaving in aspect oriented business process management. In *Conceptual Modeling - 34th International Conference, ER 2015, Stockholm, Sweden, October 19-22, 2015, Proceedings*, pages 548–557, 2015.

[85] Frédéric Jouault. Loosely coupled traceability for ATL. In *ECMDA Traceability Workshop (ECMDA-TW)*, pages 29–37, 2005.

[86] Frédéric Jouault and Ivan Kurtev. Transforming models with ATL. In *Satellite Events at the MoDELS 2005 Conference, MoDELS 2005 International Workshops, Doctoral Symposium, Educators Symposium, Montego Bay, Jamaica, October 2-7, 2005, Revised Selected Papers*, pages 128–138, 2005.

[87] Frédéric Jouault and Ivan Kurtev. On the architectural alignment of ATL and QVT. In *Proceedings of the 2006 ACM Symposium on Applied Computing (SAC), Dijon, France, April 23-27, 2006*, pages 1188–1195, 2006.

[88] Jörg Kienzle, Wisam Al Abed, Franck Fleurey, Jean-Marc Jézéquel, and Jacques Klein. Aspect-oriented design with reusable aspect models. *Trans. Aspect-Oriented Software Development*, 7:272–320, 2010.

[89] Jörg Kienzle, Wisam Al Abed, and Jacques Klein. Aspect-oriented multi-view modeling. In *Proceedings of the 8th International Conference on Aspect-Oriented Software Development, AOSD 2009, Charlottesville, Virginia, USA, March 2-6, 2009*, pages 87–98, 2009.

[90] Jacques Klein, Franck Fleurey, and Jean-Marc Jézéquel. Weaving multiple aspects in sequence diagrams. *Transactions on aspect-oriented software development*, 3:167–199, 2007.

[91] Jacques Klein, Loïc Hélouët, and Jean-Marc Jézéquel. Semantic-based weaving of scenarios. In *Proceedings of the 5th International Conference on Aspect-Oriented Software Development, AOSD 2006, Bonn, Germany, March 20-24, 2006*, pages 27–38, 2006.

[92] Anneke Kleppe, Jos Warmer, and Wim Bast. *MDA explained - the Model Driven Architecture: practice and promise*. Addison Wesley object technology series. Addison-Wesley, 2003.

[93] Alexander Knapp and Jochen Wuttke. Model checking of UML 2.0 interactions. In *Models in Software Engineering, Workshops and Symposia at MoDELS 2006, Genoa, Italy, October 1-6, 2006, Reports and Revised Selected Papers*, pages 42–51, 2006.

[94] Hans J. Köhler, Ulrich Nickel, Jörg Niere, and Albert Zündorf. Integrating UML diagrams for production control systems. In *Proceedings of the 22nd International Conference on on Software Engineering, ICSE 2000, Limerick Ireland, June 4-11, 2000.*, pages 241–251, 2000.

[95] Mirco Kuhlmann and Martin Gogolla. Strengthening sat-based validation of UML/OCL models by representing collections as relations. In *Modelling Foundations and Applications - 8th European Conference, ECMFA 2012, Kgs. Lyngby, Denmark, July 2-5, 2012. Proceedings*, pages 32–48, 2012.

[96] Jochen Malte Küster, Christian Gerth, Alexander Förster, and Gregor Engels. A tool for process merging in business-driven development. In *Proceedings of the Forum at the CAiSE'08 Conference, Montpellier, France, June 18-20, 2008*, pages 89–92, 2008.

[97] Marcello La Rosa, Marlon Dumas, Reina Uba, and Remco Dijkman. Merging business process models. *On the Move to Meaningful Internet Systems: OTM 2010*, pages 96–113, 2010.

[98] Youness Laghouaouta, Adil Anwar, Mahmoud Nassar, and Bernard Coulette. A graph based approach to trace models composition. *JSW*, 9(11):2813–2822, 2014.

[99] Agnes Lanusse, Yann Tanguy, Huascar Espinoza, Chokri Mraidha, Sebastien Gerard, Patrick Tessier, Remi Schnekenburger, Hubert Dubois, and François Terrier. Papyrus UML: an open source toolset for MDA. In *Proc. of the Fifth European Conference on Model-Driven Architecture Foundations and Applications (ECMDA-FA 2009)*, pages 1–4, 2009.

[100] Hongzhi Liang, Zinovy Diskin, Jürgen Dingel, and Ernesto Posse. A general approach for scenario integration. In *Model Driven Engineering Languages and Systems, 11th International Conference, MoDELS 2008, Toulouse, France, September 28 - October 3, 2008. Proceedings*, pages 204–218, 2008.

[101] Tianhai Liu, Michael Nagel, and Mana Taghdiri. Bounded program verification using an SMT solver: A case study. In *Fifth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012, Montreal, QC, Canada, April 17-21, 2012*, pages 101–110, 2012.

[102] Mass Soldal Lund and Ketil Stølen. A fully general operational semantics for UML 2.0 sequence diagrams with potential and mandatory choice. In *FM 2006: Formal Methods, 14th International Symposium on Formal Methods, Hamilton, Canada, August 21-27, 2006, Proceedings*, pages 380–395, 2006.

[103] Yogesh S. Mahajan, Zhaohui Fu, and Sharad Malik. Zchaff2004: An efficient SAT solver. In *Theory and Applications of Satisfiability Testing, 7th International Conference, SAT 2004, Vancouver, BC, Canada, May 10-13, 2004, Revised Selected Papers*, pages 360–375, 2004.

[104] Jacqueline A. McQuillan and James F. Power. A metamodel for the measurement of object-oriented systems: An analysis using alloy. In *First International Conference on Software Testing, Verification, and Validation, ICST 2008, Lillehammer, Norway, April 9-11, 2008*, pages 288–297, 2008.

[105] Dulani Meedeniya. *Correct model-to-model transformation for formal verification*. PhD thesis, University of St Andrews, 2013.

[106] Zoltán Micskei and Hélène Waeselynck. The many meanings of UML 2 sequence diagrams: a survey. *Software and System Modeling*, 10(4):489–514, 2011.

[107] Russ Miles and Kim Hamilton. *Learning UML 2.0 - a pragmatic introduction to UML*. O'Reilly, 2006.

[108] Brice Morin, Jacques Klein, Jörg Kienzle, and Jean-Marc Jézéquel. Flexible model element introduction policies for aspect-oriented modeling. In *Model Driven Engineering Languages and Systems - 13th International Conference, MODELS 2010, Oslo, Norway, October 3-8, 2010, Proceedings, Part II*, pages 63–77, 2010.

[109] Farida Mostefaoui and Julie Vachon. Design-level detection of interactions in aspect-uml models using alloy. *Journal of Object Technology*, 6(7):137–165, 2007.

[110] Tadao Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[111] Shin Nakajima and Tetsuo Tamai. Lightweight formal analysis of aspect-oriented models. In *UML2004 Workshop on Aspect-Oriented Modeling*, 2004.

[112] Shiva Nejati. *Behavioural model fusion*. PhD thesis, University of Toronto, 2008.

[113] Shiva Nejati, Mehrdad Sabetzadeh, Marsha Chechik, Steve M. Easterbrook, and Pamela Zave. Matching and merging of statecharts specifications. In *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007*, pages 54–64, 2007.

[114] Jaideep Nijjar and Tevfik Bultan. Unbounded data model verification using SMT solvers. In *IEEE/ACM International Conference on Automated Software Engineering, ASE'12, Essen, Germany, September 3-7, 2012*, pages 210–219, 2012.

[115] OMG. Meta Object Facility (MOF) Core v.2.4.1. Document id: formal/2013-06-01. Available at: `http://www.omg.org`. [Accessed 11 September 2013].

[116] OMG. Uml: Superstructure. v. 2.4.1. Document id: formal/2011-08-06. Available at: `http://www.omg.org`. [Accessed 22 January 2012].

[117] OMG. XML Metadata Interchange (XMI), v2.4. Document id: ptc/2010-12-06. Available at: `http://www.omg.org`. [Accessed 22 February 2012].

[118] Alloy Analyser [online]. Available from: URL: `http://www.alloy.mit.edu`. [Accessed 3 March 2012].

[119] ArcStyler 4.0 [online]. Available at: `http://www.arcstyler.com/`. [Accessed 22 February 2013].

[120] Argo UML [online]. Available at: `http://http://argouml.tigris.org/`. [Accessed 22 February 2013].

[121] Eclipse [online]. Uml2 webpage. Available at: https://eclipse.org/modeling/mdt/?project=uml2. [Accessed 1 February 2013].

[122] Kermeta [online]. Triskell metamodelling kernel. Available at: http://www.kermeta.org/, 2013. [Accessed 22 March 2011].

[123] OptimalJ [online]. Available at: http://technology.amis.nl/2006/10/22/optimalj-tutorial-for-dummies/. [Accessed 10 September 2013].

[124] RStudio [online]. Available at: https://www.rstudio.com/. [Accessed 1 February 2015].

[125] SiTra:Simple Transformer [online]. Available at: http://www.cs.bham.ac.uk/bxb/Si-tra/index.html. [Accessed 1 January 2013].

[126] Richard F Paige, Gøran K Olsen, Dimitrios S Kolovos, Steffen Zschaler, and Christopher Power. Building model-driven engineering traceability. In *ECMDA Traceability Workshop (ECMDA-TW)*, page 49, 2008.

[127] Krzysztof Pancerz and Zbigniew Suraj. Synthesis of petri net models: A rough set approach. *Fundam. Inform.*, 55(2):149–165, 2003.

[128] Tom Pender. *UML bible*. John Wiley & Sons, Inc., 2003.

[129] Gilles Perrouin, Gilles Vanwormhoudt, Brice Morin, Philippe Lahire, Olivier Barais, and Jean-Marc Jézéquel. Weaving variability into domain metamodels. *Software and System Modeling*, 11(3):361–383, 2012.

[130] Bernhard Pfahringer. Conjunctive normal form. In *Encyclopedia of Machine Learning*, pages 209–210. 2010.

[131] Laleh Pirzadeh. Human factors in software development: a systematic literature review. Master's thesis, Chalmers University of Technology, 2010.

[132] Rachel Pottinger and Philip A. Bernstein. Merging models based on given correspondences. In *Proceedings of the 29th international conference on Very large data bases-Volume 29*, pages 826–873, 2003.

[133] Raghu Reddy, Arnor Solberg, Robert France, and Sudipto Ghosh. Composing sequence models using tags. In *Proceedings of MoDELS workshop on Aspect Oriented Modeling, Genova, Italy*, 2006.

[134] Marcello La Rosa, Marlon Dumas, Reina Uba, and Remco M. Dijkman. Merging business process models. In *On the Move to Meaningful Internet Systems: OTM 2010 - Confederated International Conferences: CoopIS, IS, DOA and ODBASE, Hersonissos, Crete, Greece, October 25-29, 2010, Proceedings, Part I*, pages 96–113, 2010.

[135] Julia Rubin, Marsha Chechik, and Steve M. Easterbrook. Declarative approach for model composition. In *International Workshop on Modeling in Software Engineering, MiSE 2008, Leipzig, Germany, May 10-11, 2008*, pages 7–14, 2008.

[136] Ragnhild Runde. *STAIRS - understanding and developing specications expressed as UML interaction diagrams*. PhD thesis, University of Oslo, 2007.

[137] Johannes Sametinger. *Software engineering with reusable components*. Springer, 1997.

[138] Vladimiro Sassone, Mogens Nielsen, and Glynn Winskel. Models for concurrency: Towards a classification. *Theoretical Computer Science*, 170(1-2):297–348, 1996.

[139] Seyyed Shah, Kyriakos Anastasakis, and Behzad Bordbar. Using traceability for reverse instance transformations with sitra. *Design and Architectures for Signal and Image Processing (DASIP 2008). Special Session on Formal Models, Transformations and Architectures for Re liable Embedded System Design, Bruxelles, Belgium*, 2008.

[140] Ilya Shlyakhter, Robert Seater, Daniel Jackson, Manu Sridharan, and Mana Taghdiri. Debugging overconstrained declarative models using unsatisfiable cores. In *Automated*

Software Engineering, 2003. Proceedings. 18th IEEE International Conference on, pages 94–105. IEEE, 2003.

[141] Mathias Soeken, Robert Wille, and Rolf Drechsler. Verifying dynamic aspects of uml models. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011*, pages 1–6. IEEE, 2011.

[142] John Spivey and JR Abrial. *The Z notation*. Prentice Hall Hemel Hempstead, 1992.

[143] Thomas Stahl, Markus Völter, Jorn Bettin, Arno Haase, and Simon Helsen. *Model-driven software development - technology, engineering, management*. Pitman, 2006.

[144] Harald Störrle. A petri-net semantics for sequence diagramms. In *FBT*, pages 233–242, 1999.

[145] Harald Störrle. Trace semantics of interactions in uml 2.0. *J. Visual Languages and Computing*, 2004.

[146] Emina Torlak and Daniel Jackson. Kodkod: A relational model finder. In *Tools and Algorithms for the Construction and Analysis of Systems, 13th International Conference, TACAS 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007 Braga, Portugal, March 24 - April 1, 2007, Proceedings*, pages 632–647, 2007.

[147] Jan Tretmans. Model based testing with labelled transition systems. In *Formal Methods and Testing, An Outcome of the FORTEST Network, Revised Selected Papers*, pages 1–38, 2008.

[148] Sebastian Uchitel, Greg Brunet, and Marsha Chechik. Synthesis of partial behavior models from properties and scenarios. *Software Engineering, IEEE Transactions on*, 35(3):384–406, 2009.

[149] Robert Valette. Analysis of petri nets by stepwise refinements. *J. Comput. Syst. Sci.*, 18(1):35–46, 1979.

[150] Fernando Valles-Barajas. Use of a lightweight formal method to model the static aspects of state machines. *ISSE*, 5(4):255–264, 2009.

[151] Jon Whittle, João Araújo, and Ana Moreira. Composing aspect models with graph transformations. In *Proceedings of the 2006 international workshop on Early aspects at ICSE*, pages 59–65. ACM, 2006.

[152] Jon Whittle and Praveen K. Jayaraman. Synthesizing hierarchical state machines from expressive scenario descriptions. *ACM Transactions on Software Engineering and Methodology*, 19(3):8:1–8:45, February 2010.

[153] Jon Whittle, Duminda Wijesekera, and Mark Hartong. Executable misuse cases for modeling security concerns. In *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008*, pages 121–130, 2008.

[154] Magdalena Widl, Armin Biere, Petra Brosch, Uwe Egly, Marijn Heule, Gerti Kappel, Martina Seidl, and Hans Tompits. Guided merging of sequence diagrams. In *Software Language Engineering, 5th International Conference, SLE 2012, Dresden, Germany, September 26-28, 2012, Revised Selected Papers*, pages 164–183, 2012.

[155] Manuel Wimmer, Andrea Schauerhuber, Gerti Kappel, Werner Retschitzegger, Wieland Schwinger, and Elisabeth Kapsammer. A survey on UML-based aspect-oriented design modeling. *ACM Comput. Surv.*, 43(4):28, 2011.

[156] Murray Woodside, Dorina C Petriu, Dorin B Petriu, Jing Xu, Tauseef Israr, Geri Georg, Robert France, James M Bieman, Siv Hilde Houmb, and Jan Jürjens. Performance analysis of security aspects by weaving scenarios extracted from uml models. *Journal of Systems and Software*, 82(1):56–74, 2009.

[157] Chunhua Yang. Detecting and evaluating semantic influences of aspect weaving in aspect oriented models. *JSW*, 8(11):2675–2681, 2013.

[158] Ingrid Chieh Yu and Henning Berg. A framework for metamodel composition and adaptation with conformance-preserving model migration. In *Model-Driven Engineering and Software Development - Third International Conference, MODELSWARD 2015, Angers, France, February 9-11, 2015, Revised Selected Papers*, pages 133–154, 2015.

[159] D. Zhang, S. Li, and X. Liu. An approach for model composition and verification. In *NCM 2009*, pages 1102–1107. IEEE Computer Society Press., 2009.

[160] MengChu Zhou, Frank DiCesare, and Alan A Desrochers. A hybrid methodology for synthesis of petri net models for manufacturing systems. *Robotics and Automation, IEEE Transactions on*, 8(3):350–361, 1992.

[161] MengChu Zhou, Kevin McDermott, and Paresh A. Patel. Petri net synthesis and analysis of a flexible manufacturing system cell. *Systems, Man and Cybernetics, IEEE Transactions on*, 23(2):523–531, 1993.