SCALABLE FORMAL VERIFICATION OF UML MODELS

Doctoral Dissertation of:
Mohammad Mehdi Pourhashem Kallehbasti

Supervisor:
Prof. Luciano Baresi

Tutor:
Prof. Carlo Ghezzi

The Chair of the Doctoral Program:
Prof. Carlo Fiorini

2015 – XXVIII
Acknowledgement

My special thanks goes to my supervisor, Prof. Luciano Baresi, and Prof. Matteo Rossi for three years of unconditional support, precious advices, and unforgettable kindness. They always have been generous to share their time and expertise with me.

I am really grateful to all without whom it would be difficult, if not impossible, to start and accomplish this PhD: Prof. Carlo Ghezzi, Drs. Amir Molzam Sharifloo, Alfredo Motta, Marcello Bersani, Srdan Krstic, and Ramin Amouchie.

I would also like to thank Dr. Marco Roveri for accepting to review my manuscript and giving valuable comments.
To my father for his lifelong support
and
to my mother for her immeasurable love.
Abstract

THE Unified Modeling Language (UML) is a general-purpose platform-independent modeling language, created and managed by the Object Management Group (OMG). During recent years, UML has transformed from a relatively basic descriptive tool to a sophisticated prescriptive one that can be used to specify, analyze and implement complex software systems. It offers a heterogeneous set of diagrams to describe the different views of a software system during the Model-Driven Development (MDD) process.

Model defects are a significant concern in the MDD paradigm, as model transformations and code generation may propagate errors to other notations where they are harder to detect and trace. To avoid this problem, formal verification techniques are used, that can check the correctness of models and verification results often reveal unexpected behaviors in the models, which if left unresolved, may lead to a tremendous cost in the later stages of the development.

UML notations usually come with a reasonably well-defined syntax, but their semantics are left underspecified and open to different interpretations. This freedom hampers the formal verification of produced specifications and calls for more rigor and precision. This thesis aims to bridge this gap and proposes a flexible and modular formalization approach based on temporal logic. We studied the different interpretations for some of its constructs, and our framework allows one to assemble the semantics of interest by composing the selected formalizations for the different pieces.

However, the formalization per se is not enough. The verification process, in general, becomes slow and impossible –as the model grows in size. To tackle the scalability problem, this thesis also proposes an efficient bit-vector-based encoding of LTL/CLTL formulae. The experimental results witness a significant increase in the size of analyzable models, not only for our formalization of UML models, but also for numerous other models that can be reduced to bounded satisfiability checking of LTL/CLTL formulae.

This thesis contributes towards formal specification of UML/eFT-UML (extensible Formal, Timed UML) models by flexible modular semantics implemented in Corretto, our verification tool, that covers a significant number of diagram types in a single coherent framework. Moreover, the proposed bit-vector-based encodings are implemented as
plugins in Zot Bounded Model/Satisfiability Checker to tackle the scalability problem of Corretto.
## Contents

1 Introduction
   1.1 Problem Formulation .................................................. 2
   1.2 Contributions .......................................................... 5
   1.3 Dissertation Roadmap .................................................. 6

2 The Temporal Satisfiability Checking problem ........................................ 7
   2.1 Boolean Satisfiability Problem ........................................ 8
   2.2 Satisfiability Modulo Theories ......................................... 8
      2.2.1 Theories .......................................................... 9
   2.3 Linear Temporal Logic .................................................. 9
   2.4 Bounded Satisfiability Checking ....................................... 11
   2.5 BSC for LTL with Eventualities ..................................... 12
   2.6 Constraint LTL (CLTL) and CLTL Over Clocks (CLTLoc) ............... 15
   2.7 TRIO Metric Temporal Logic .......................................... 17
   2.8 Related Work .......................................................... 20

3 eFT-UML
   3.1 Structure .............................................................. 22
   3.2 Time Dimension ......................................................... 23
   3.3 Syntax and Semantics .................................................. 24
      3.3.1 Class Diagram .................................................. 25
      3.3.2 State Machine .................................................. 27
      3.3.3 Sequence Diagram ............................................. 29
      3.3.4 Shared Events ................................................ 30
   3.4 Related Work .......................................................... 32

4 Flexible modular formalization of UML models ....................................... 35
   4.1 SD and Combined Fragments .......................................... 36
   4.2 Semantic Variation Points and Variants ................................ 40
   4.3 Basic Modules .......................................................... 43
      4.3.1 Order .......................................................... 43
Contents

4.3.2 Borders ................................................. 44
4.3.3 Auxiliary Operators ................................. 45
4.4 Combined Fragments .................................. 45
4.5 Integrated Modular Semantics ....................... 49

5 Bit-vector-based Encoding for LTL and CLTL 51
5.1 Quantifier-Free fixed-size Bit-vector Logic ........ 51
5.2 Bit-vector-based LTL Encoding Setup .............. 53
5.3 Bit-vector-based Encoding ............................ 53
5.4 Correctness Proof .................................... 58
5.5 Combining Bit-vectors and Arithmetic to solve CLTL 60

6 Tools and Experimental Evaluation .................... 63
6.1 Corretto .............................................. 63
6.2 Experimental Evaluation for Formal SDs .......... 64
6.3 Zot .................................................. 67
6.4 Experimental Evaluation for bvzot .................. 68
6.5 Experimental Evaluation for ae2bvzot .............. 75

7 Conclusion ................................................. 87
7.1 Reviewing Research Questions ....................... 87
7.2 Limitations and Future Work ......................... 89

Appendices ................................................. 91
A Appendix ................................................. 93
A.1 CF_Alt ............................................... 93
A.2 CF_Opt ........................................... 95
A.3 CF_Par ............................................. 96
A.4 CF_Loop ........................................... 97
A.5 CF_Break ........................................ 100
A.6 Combine ........................................... 102
A.7 Link_Pre_Post ...................................... 103

Bibliography ............................................. 105
# List of Figures

1.1 Problem formulation. ........................................ 4

2.1 An infinite, ultimately periodic execution with loop at position \( l \). .... 12

3.1 Structure of valid \texttt{eFT-UML} model. .......................... 22
3.2 Class notation. .................................................. 26
3.3 Clocks meta-model in \texttt{eFT-UML}. .............................. 26
3.4 An example Class Diagram. ....................................... 27
3.5 An example State Machine Diagram. .............................. 27
3.6 Meta-model of transitions in \texttt{eFT-UML} State Machines. .... 28
3.7 3.7a: Syntax of transitions in \texttt{eFT-UML} SMs. 3.7b: An example of a transition. ........................................ 28
3.8 An example Sequence Diagram. .................................... 30
3.9 The transition from State0 to State1 of Figure 3.9b enables the found message of Figure 3.9a. The message of Figure 3.9a triggers the transition from State1 to State0 of 3.9b. ........................................ 31

4.1 Meta-model of Combined Fragments in a Sequence Diagram. .......... 36
4.2 An example Sequence Diagram and its parts. .......................... 37
4.3 An example Alternative Combined Fragment. .......................... 38
4.4 An example Optional Combined Fragment. ............................ 38
4.5 An example Parallel Combined Fragment. ............................... 39
4.6 An example Loop Combined Fragment. .................................. 39
4.7 An example Break Combined Fragment. .................................. 40
4.8 Example Sequence Diagram. ........................................ 41
4.9 Different configurations obtained from the four choices. .............. 43

5.1 Calculation of the formula \((\psi_1S\psi_2)SY\phi_3\). ....................... 56

6.1 SDSearch Sequence Diagram. ..................................... 65
6.2 Overall architecture of \texttt{Zot} ..................................... 68
6.3 Time/Memory Comparison for KRC1 (SAT and P1). ..................... 70
List of Figures

6.4 Time/Memory Comparison for KRC1 (P2) ........................................ 71
6.5 Time/Memory Comparison for KRC2 (SAT and P1) ........................ 71
6.6 Time/Memory Comparison for KRC2 (P2) ........................................ 72
6.7 Time/Memory Comparison for Fischer (SAT and P1) ...................... 72
6.8 Time/Memory Comparison for SDServerl2 (SAT and P1) ............ 73
6.9 Time/Memory Comparison for SDServerl3 (SAT and P1) ............ 73
6.10 Comparison between classic and bit-vector-based encoding of LTL formulae .......................................................... 74
List of Tables

2.1 Loop constraints. .................................................. 12
2.2 Last state constraints. .......................................... 13
2.3 Propositional subformulae constraints. ...................... 13
2.4 Future temporal subformulae constraints. .................... 14
2.5 Eventuality constraints. ........................................ 14
2.6 Past temporal subformulae constraints. ...................... 14
2.7 First state constraints. ......................................... 15
2.8 TRIO temporal operators ........................................ 18
2.9 Variants of TRIO temporal operators ......................... 19

5.1 Signature of used QF_BV operators. .......................... 52
5.2 Definition of auxiliary operators. ............................. 53
5.3 QF_BV encoding of propositional formulae. ................. 54
5.4 QF_BV encoding of basic temporal operators. ............... 54
5.5 An example of calculation of bit-vector for \( S \). ............ 55
5.6 Definition of additional auxiliary operators. ................. 57
5.7 QF_BV encoding of derived temporal operators. ............ 58

6.1 Experimental results with Corretto (times are in seconds). . 66
6.2 Comparison between \texttt{ae2zot} and \texttt{ae2bvzot} on the Sorting models. 79
6.3 Comparison between \texttt{ae2zot} and \texttt{ae2bvzot} on the CCAS models. 82
6.4 Comparison between \texttt{ae2zot} and \texttt{ae2bvzot} on the Leader models. 83
6.5 Comparison between \texttt{ae2zot} and \texttt{ae2bvzot} on the UML-Leader models. 84
6.6 Comparison between \texttt{ae2zot} and \texttt{ae2bvzot} on the Lamp models. 84
6.7 Comparison between \texttt{ae2zot} and \texttt{ae2bvzot} on the Counting models. 85
6.8 Comparison between \texttt{ae2zot} and \texttt{ae2bvzot} on the Spikes and Wave models. 86
CHAPTER 1

Introduction

“It is a privilege to be software developer, because we change the world. It is a responsibility to be software developer, because we change the world.”
Grady Booch

Computer systems are evolving rapidly. The first computers were highly dependent on human intervention to accomplish a given task. Strictly speaking, human was completing computers. After decades of exponential growth of computers, they are everywhere and play important roles in our daily lives, such that we are highly dependent on computers, and in some cases, we no longer live without embedded medical devices in our bodies. In other words, nowadays, the contrary of what was said earlier is true and computer completes human.

Computer engineers are evolving both software and hardware towards satisfying increasing needs of the world. It rapidly changes the world, which is both fascinating and threatening at the same time. Among concerns directly or indirectly caused by computer systems like privacy, confidentiality, security, and many failures that cost billions of dollars, human safety is the most important one. Therac-25, a radiation therapy machine that gave overdoses of radiation and killed three patients, is an example of software failure, with irretrievable consequences. Hence, in software development, in general, in safety critical systems in particular, the guarantee of safety is inevitable. Examples of such safety critical systems are medical devices, nuclear power plants, avionics, production lines, and many others, where safety can be warranted by means of formal methods. Once a system is formally specified, we can prove that desired requirements are ensured and superfluous behaviors never happen under any circumstances.

The remainder of this introductory chapter is organized as follows. Section 1.1 renders the problems being investigated in this thesis. Next, in Section 1.2, the contributions of this thesis is presented. Finally, Section 1.3 gives a brief introduction to chapters of this
Chapter 1. Introduction

document.

1.1 Problem Formulation

Today’s software systems are huge and complex, therefore abstracting and modeling are used to reduce the systems complexity and pave the way for efficient software development. Model-Driven Engineering (MDE) [52] is a software development methodology that has gained significant interest over past years. MDE simplifies the design process, boosts productivity, and promotes communication between individuals and teams by standardization of terminology and models.

The Unified Modeling Language (UML) is a general-purpose, platform-independent modeling language, created and developed by G. Booch et al. [17]. UML was adopted as a standard and has been managed by the Object Management Group (OMG). During Model-Driven Development (MDD) [76] process UML is used that provides a significant set of diagrams to describe static (structural) and dynamic (behavioral) aspects of the system. UML models, as artifacts of the design phase, demonstrate the prospective software system at an abstract level.

Model defects are a significant concern as model transformations and code generation may amplify and propagate errors to other notations where they are harder to detect and trace. Rigorous specifications allow the designer to reason on the models and detect defects and inconsistencies early in the specification process without propagating and amplifying them further. This can be done using model checking, a formal verification technique, where the system is formally specified and the expected behavior (property) is usually expressed in temporal logic. Essentially, we are trying to find a scenario, where our formalization of the system, neglects/violates the property of interest. In this technique, state space of the system is exhaustively explored to find a counter example, more precisely $\text{System} \land \neg\text{Property}$. If such a trace is not found, the property holds for the system, otherwise the found trace guides towards the source of model defect(s).

UML models, besides serving their purposes mentioned earlier, can be used as input for Model checking techniques, such that by means of a verification tool, designers can verify correctness of the designed model, even when it is not completely describing the whole system yet. It helps designers to design and verify software products in an incremental manner and detect inconsistencies as early as possible.

Throughout the thesis, we focus on two problems that are rendered as the following research questions:

RQ 1. How to formally specify diverse UML diagrams as a single integrated model that captures all possible behaviors of a system?

There have been many different attempts to ascribe the notation with a (more) formal semantics, but the wideness of the language has often led the authors to only concentrate on some diagram types, while neglecting the key characteristics of UML, that is, the rich set of diagram types and the freedom with which the designer can model a system. The importance of this issue arises from the fact that the most unexpected and critical errors are due to inconsistencies among different diagrams. Different UML diagrams, like Class Diagram (CD), Object Diagram (OD), Interaction Overview Diagram (IOD), Sequence Diagram (SD), and State Machine (SM) have been formalized using several formalisms, such that the formal model is very similar to the original model, and there
1.1. Problem Formulation

is a convenient way of mapping elements in the model to the corresponding elements in the formal model. For example, Labeled Transition System (LTS) is used for state machines and Petri nets for activity diagrams.

These formalizations that aim a specific type of diagram, fail to capture the behavior of system from different points of view, consequently orthogonal view of the system. Different diagram types focus on different aspects of the system. For example state machines monitor states of important objects in the system, while, sequence diagrams capture the behavior of objects interacting with one another in order to fulfill the functionality of the system. Supporting a set of heterogeneous behavioral views is desirable to meet industry requirements, like software life-cycle, documentation and code generation. For the sake of performing a comprehensive formal verification, all these views must be regarded, ideally notwithstanding how an organization handles its models.

Formal verification is done in two main steps. The first step is to formally specify the system at an abstraction level using a formalism. The property, usually expressed in temporal logics like LTL (Linear Temporal Logic [51]), in turn is checked against the formal model. Two important factors in the formal specification are expressivity of the underline formalism and its decision procedures efficiency. The system to be formalized is already an abstracted model, hence the underline formalism must be expressive enough in order to enable the formal model to capture behaviors of such system with a reasonable precision. Since LTL is well suited for capturing the dynamic aspects of UML models and offers the required flexibility for their formalization, we continue exploring the problems regarding LTL formalism and LTL-derived logics. Moreover, another reason of LTL being favorable in this research area is that, temporal logic based specifications can benefit from symbolic model checking techniques, in which states explosion problem is not as crucial as it is in explicit-state model checking.

RQ 1.1. How to formalize those UML elements with many meanings, without losing generality?

As a semi-formal language, UML has a formal syntax and an informally defined semantics. In fact, its semantics is intentionally underspecified to provide leeway for domain-specific refinements. On the other hand, general semantics hampers formal verification owing to the lack of required precision. The behavior of the modeling elements must be stated unambiguously, whereas informal semantics of UML diagrams result in different interpretations. These ambiguities mainly rise from sequence diagrams, which are often used to capture the most significant scenarios that describe how the components of a complex system interact. Focusing on sequence diagrams, many researchers propose their interpretations, among which only few [37, 53] present formalization amenable for formal verification. Although, their formalizations are based on fixed semantics with limited interoperability.

RQ 2. How to formally verify bigger UML models or generally any model specified in LTL?

The second step towards formal verification is to use a verification technique to verify the formally specified model, while the state explosion has always been a crucial problem. With respect to the problem under consideration, where the formal verification
Chapter 1. Introduction

Figure 1.1: Problem formulation.

of UML models is reduced to satisfiability checking of LTL formulae, as the UML model becomes more complex, the resulting LTL formula grows in size accordingly and makes the whole verification process slow, and at a given point impossible.

Complete and Bounded Satisfiability Checking (BSC) [71] are the two main techniques towards checking whether or not an LTL formula is satisfiable. In BSC, LTL formulae are suitably translated into formulae of another decidable logic, such as (Propositional) Boolean Logic, which precisely capture ultimately periodic models of the original formulae of length up to a bound $K$; produced formulae are then fed to a solver (e.g., a SAT solver) for verification. BSC is the fastest approach to reveal system errors, but to make it complete and verify properties, a big enough $K$ must be determined by algorithms, that are expensive [11]. However, complete approaches suffer much more from the state explosion problem and bounded approaches are preferred specially when complete ones give up in returning verification result for huge formal models. Still, scalability is a problem even in bounded approaches, that hampers verification of big
1.2 Contributions

The two parallel contributions of this thesis aim to drive research questions on the formal specification and scalable verification of eFT-UML (extensible Formal, Timed UML) models. eFT-UML integrates a significant number of diagram types in a single coherent framework including CD, OD, IOD, SM, and SD. The goal of eFT-UML is to overcome the limitation of separated diagram types by means of a precise set of shared events. In addition, it borrows from MARTE [66] the notion of time. This means that eFT-UML is particularly suited for the specification of timed systems, that is, systems whose correctness is not just functional, but it also depends on the delays associated with the different activities and with the temporal ordering of the different events. The formal semantics of eFT-UML is based on the TRIO [22] metric temporal logic, which gives us the flexibility and composability required to specify the semantics of a complex notation.

1. Flexible modular formalization of eFT-UML models

The first contribution is a flexible modular formalization for eFT-UML elements with many meanings, that mainly belong to SD. We studied the most significant semantics proposals, organized them into a single coherent framework, and proposed a solution to interpret SD in a compositional and modular way, in order to fulfill OMG's ambition of keeping UML useful in many domains. Users can decide the interpretations of the key aspects of their interest and the result is a complete and coherent semantics; then, provided some simple constraints are respected to avoid making inconsistent decisions, our framework accommodates all other aspects.

The proposed theoretical approach is implemented on top of our verification toolset, namely Corretto¹, where the produced temporal logic formula is fed to Zot² our bounded model/satisfiability checker, allows the user to easily play with the different semantics. The user can simulate the behavior of the diverse semantics and verify the satisfiability of the properties of interest. S/he can thus understand how the system behaves or work on the satisfiability of the properties and then obtain the guarantees the system must offer. The same theoretical approach can be used for SM.

2. Efficient scalable encoding for LTL and CLTL formulae

While working on a solution for making the verification of UML models scale properly, we have identified an interesting solution that suites a wider class of verification problems that can be reduced to TRIO, therefore Constraint LTL (CLTL)/LTL formulae. The second contribution of this thesis is a bit-vector-based encoding of these CLTL/LTL formulae, which has allowed us to move a significant step forward in tackling the scalability of their formal verification. The

¹https://github.com/deib-polimi/Corretto/
²https://github.com/fm-polimi/zot
proposed encoding for LTL and CLTL are implemented as plugins entitled bvzot and ae2bvzot in Zot our satisfiability/model checker.

1.3 Dissertation Roadmap

This thesis provides background for eFT-UML and LTL encoding assuming reader to be familiar with standard UML and basics of formal methods. The remainder of the thesis is structured as following.

Chapter 2 – The Temporal Satisfiability Checking problem.
Required background for BSC of LTL formulae and the classic Boolean-based LTL encoding are presented in this chapter, together with Constraint LTL (CLTL) and TRIO metric temporal logic.

Chapter 3 – eFT-UML.
This chapter gives an introduction to eFT-UML diagrams, their formal semantics, and our verification tool Corretto that automatically produces the formal semantics.

Chapter 4 – Flexible modular formalization of UML models.
This chapter presents the first contribution of this thesis, that is a flexible modular formalization of UML models that accommodates all the existing formal semantics.

Chapter 5 – Bit-vector-based Encoding for LTL and CLTL.
The second contribution of this thesis is presented in this chapter, that is novel bit-vector-based encodings for LTL and CLTL formulae. We present proof of correctness of the encodings, and also explain why the novel encoding for LTL is faster and more scalable than all other encodings and why CLTL encoding is mostly more efficient than the other tool.

Chapter 6 – Tools and Experimental Evaluation.
This chapter reports the experimental evaluation related to our contributions, in order to discuss how the flexible modular semantics is useful and also compare the novel encodings against the state of the art tools in the field.

Chapter 7 – Conclusion.
Finally, this chapter concludes the work presented in this thesis by explaining how the research questions are addressed and presenting limitations of the proposed approaches and future work.
Linear Temporal Logic (LTL) and its variants have been used in computer science for decades [69]. Its applications include the specification and verification of (possibly safety-critical) programs and systems [24–26, 72], test case generation [80], run-time verification [9], planning [36], and controller synthesis [79]. In addition, LTL —or its expressively equivalent variants— can be used as underlying formalism to capture the semantics of semi-formal notations like UML to perform formal verification on them, as we showed in [7].

However, for an LTL-based approach to be effective in practice, there must be techniques and tools that allow users to check big specifications in a short amount of time. In this chapter, we focus on the problem of performing formal verification on systems described through a set of LTL formulae. These formal descriptions could be the outcome of systematic requirements elicitation or formal specification, but they could also be the output of tools producing LTL formalizations from more informal languages such as UML.

Formal verification of LTL specifications can be carried out through tools capable of determining the satisfiability of sets of LTL formulae. Various techniques have been developed in the past, complete approaches that are based on automata construction, and bounded approaches that analyze the system within a limited number of steps; both of which come with pros and cons.

The chapter is structured as follows. Sections 2.1-2.2 introduce the Boolean satisfiability problem and satisfiability modulo theories. Sections 2.3-2.4 introduce LTL and Bounded Satisfiability Checking (BSC) approaches, and reason about why BSC approaches are more favorable in many cases. The classic bounded encoding of LTL formula based on eventuality is presented in details in Section 2.5. Constraint LTL (CLTL) and CLTL over clocks are introduced in Section 2.6. Since the formalization of
Chapter 2. The Temporal Satisfiability Checking problem

eFT-UML is in TRIO metric temporal logic, that is the first contribution of this thesis, we introduce this logic in Section 2.7. Finally, Section 2.8 surveys number of the most related works.

2.1 Boolean Satisfiability Problem

Boolean satisfiability (SAT) is the problem of determining whether or not there exists a variable assignment that evaluates the given Boolean formula to true. For example, \( a \land \neg b \) is satisfiable because there exists a variable assignment (\( a \) is true, and \( b \) is false) that makes the given formula to be evaluated to true. In contrast, \( a \land \neg a \) is unsatisfiable as a result of being evaluated to false for all the possible variable assignments.

The SAT problem has many applications in electronic design automation, software and hardware verification, automatic theorem proving, and artificial intelligence. There has been outstanding progress in SAT algorithms and SAT solver technology in recent years, summarized in a survey \[81\].

A Boolean formula is defined over a set of Boolean variables \( B \). The syntax of Boolean formulae is defined by the following grammar:

\[
\phi ::= b | \neg \phi | \phi \land \phi
\]

where \( b \in B \), whose domain is \{\( \top, \bot \}\}. \( \neg \) and \( \land \) are the main connectives called “not” and “and”, respectively. The expressions \( \phi_1 \lor \phi_2 \), \( \phi_1 \Rightarrow \phi_2 \), \( \phi_1 \Leftrightarrow \phi_2 \), and \( \phi_1 \oplus \phi_2 \) are also commonly used as abbreviations for \( \neg (\neg \phi_1 \land \neg \phi_2) \), \( \neg \phi_1 \lor \phi_2 \), \( (\phi_1 \Rightarrow \phi_2) \land (\phi_2 \Rightarrow \phi_1) \), and \( \neg (\phi_1 \Leftrightarrow \phi_2) \), respectively.

Most of the SAT solvers take as input the Boolean formula represented in Conjunctive Normal Form (CNF), that is a conjunction of clauses, where each clause is a disjunction of literals, and a literal is either a variable or its negation. However, working directly on the Boolean circuit representation can be more efficient for SAT applications in circuit domain. On the other hand, most of the modern SAT solvers are based on a (Davis-Putnam-Logemann-Loveland) DPLL \[38\] procedure and its variants \[39\]. Minisat\(^1\) and zChaff\(^2\) are two well-known SAT solvers that are based on DPLL procedures.

2.2 Satisfiability Modulo Theories

Satisfiability modulo theories (SMT) generalizes SAT by combining background theories expressed in first-order logic with equality, like theories of real numbers, integers, arrays, lists and fixed-size bit-vectors. The tools that check satisfiability of formulae in these theories are called SMT solvers. The two main approaches in the SMT solvers are eager and lazy. In the eager approach theory-specific translators translate a formula into an equisatisfiable formula in propositional logic, that is similar to compilers that optimize generated code from a higher level program. On the other hand, lazy approaches build efficient theory solvers by combining SAT solvers with a decision procedure for conjunctions of theory literals \[20\]. Applications of SMT solvers include bounded model checking over infinite domains, predicate abstraction, and extended static checking \[61\].

In formal verification of complex systems, we need to check properties that are behaviors defined by objects and their interactions; and a Boolean formula is not usually

\[\text{http://minisat.se/}\]
\[\text{http://www.princeton.edu/~chaff/zchaff.html}\]
expressive enough to capture these behaviors. For example, a richer language is required to capture different aspects of a program that manipulate arrays. Combining different theories, enables us to formalize such programs, together with its property for its formal verification. The satisfiability checking of the resulting formula by combining several theories is no more a SAT problem, but a SAT modulo theories problem.

2.2.1 Theories

A $\Sigma$-theory is a set of sentences over a signature $\Sigma$. Given $T$ as a theory, if $T \cup \{\phi\}$ is satisfiable then $\phi$ is called satisfiable modulo $T$. For example, suppose that the signature $\Sigma$ contains 0, 1, $+$, $-$, and $<$, that are interpreted by $\mathbb{Z}$ structure, then the set of first-order sentences that are true in $\mathbb{Z}$, is called theory of linear arithmetic. Some of the most frequently utilized theories are summarized as follows [28].

- **Linear Arithmetic**: is the theory whose only admissible arithmetic functions are $+$ and $-$, and the functions are applied to variables or numerical constants. Multiplication is also allowed, provided that at least one operand is a numerical constant. For example, $24 \cdot x$ is legal, where $x$ is an arithmetic variable. Atomic predicates can be formed using equality and inequality ($=, <$).

- **Difference Arithmetic**: also known as separation logic or difference logic is a sub-theory of the theory of linear arithmetic, where each inequality must be in the form of $x - y \leq c$, for $x, y$ arithmetic variables and $c$ a numerical constant. For example, $x = y$ has the constraints $x - y \leq 0$ and $y - x \leq 0$.

- **Uninterpreted Functions** also known as free theory, is over a signature $\Sigma$ with an empty collection of sentences. The importance of decision procedures for this theory rises from the fact that the decision problem of many other theories, like arrays, are reducible to this theory.

  A congruence closure can be used to present the smallest set of implied equalities having equalities between terms using uninterpreted functions. The satisfiability of a combination of equalities and inequalities can be checked using this representation. For example, in the case of inequality, both sides must be in different equivalence classes.

- **Bit-vectors** The computer’s arithmetic is different from the arithmetic on mathematical integers, because of integers appearing in fixed size registers. Using bit-vectors (sequences of bits) are the most appropriate way of presenting bounded variables in the machine arithmetic. The bit-vector theory allows arithmetic operations, like binary addition and subtraction, and bit-wise operations, like conjunction and disjunction. The bit-vectors theory can easily reduced to SAT by bit-blasting bit-vector formulae. The Quantifier-Free fixed-size Bit-vector Logic (QF_BV) is introduced in Section 5.1.

2.3 Linear Temporal Logic

LTL is a widely-used specification logic. Our focus is on the variant with both future and past temporal operators. In fact, although past operators do not increase the expressiveness of the logic, they are advantageous for compositional reasoning [59].
addition, LTL with past operators is exponentially more succinct than its future-only counterpart [55]. Usually, LTL refers to the logic with only future temporal operators, however, we keep calling the mentioned logic LTL. Although, in the literature it is called PLTL (LTL with past operators) and sometimes LTLB (LTL with both past and future).

An LTL formula is defined over a set of atomic propositions $AP$. The syntax of LTL is defined by the following grammar:

$$
\phi ::= p \mid \neg \phi \mid \phi \wedge \phi \mid X\phi \mid Y\phi \mid \phi U \phi \mid \phi S \phi
$$

where $p \in AP$, $X$ and $U$ are the “next” and “until” future operators, whereas $Y$ (“yesterday”) and $S$ (“since”) are their past counterparts.

The semantics of LTL is given in terms of infinite sequences of sets of atomic propositions, or words. A word $\pi: \mathbb{N} \to 2^{AP}$ assigns, to every instant of the temporal domain $\mathbb{N}$, the (possibly empty) set of atomic propositions that hold in that instant. We can think of a word as an infinite sequence of states $\pi = s_0 s_1 s_2 \ldots$, where each state is labeled with the atomic propositions that hold in it. We say that a word satisfies formula $\phi$ at instant $i$, written $\pi, i \models \phi$, if $\phi$ holds when evaluated starting from instant $i$ of $\pi$.

The following is the usual formal semantics of the satisfiability relation for LTL:

$$
\begin{align*}
\pi, i &\models p & \iff & p \in \pi(i) \text{ for } p \in AP \\
\pi, i &\models \neg \phi & \iff & \pi, i \not\models \phi \\
\pi, i &\models \phi_1 \wedge \phi_2 & \iff & \pi, i \models \phi_1 \text{ and } \pi, i \models \phi_2 \\
\pi, i &\models X\phi & \iff & \pi, i + 1 \models \phi \\
\pi, i &\models Y\phi & \iff & i > 0 \text{ and } \pi, i - 1 \models \phi \\
\pi, i &\models Z\phi & \iff & i = 0 \text{ or } i > 0 \text{ and } \pi, i - 1 \models \phi \\
\pi, i &\models \phi_1 U \phi_2 & \iff & \exists j \geq i \text{ s.t. } \pi, j \models \phi_2 \\
& & \text{and } \forall n \text{ s.t. } i \leq n < j : \pi, n \models \phi_1 \\
\pi, i &\models \phi_1 S \phi_2 & \iff & \exists j \leq i \text{ s.t. } \pi, j \models \phi_2 \\
& & \text{and } \forall n \text{ s.t. } j < n \leq i : \pi, n \models \phi_1
\end{align*}
$$

We say that a word $\pi$ satisfies LTL formula $\phi$ when it holds in the first instant of the temporal domain, i.e., when $\pi, 0 \models \phi$. In this case we will sometimes write $\pi \models \phi$. A word $\pi$ that satisfies $\phi$ is a model for $\phi$.

Starting from the basic connectives and operators, it is customary to introduce the other traditional Boolean connectives ($\lor$, $\Rightarrow$, ...), and temporal operators as abbreviations. In particular the “eventually in the future” (F), “globally in the future” (G) and “release” (R) operators (and their past counterparts “eventually in the past” P, “historically” H and “trigger” T) are defined as follows:

$$
\begin{align*}
F\phi &= \top U \phi & P\phi &= \top S \phi \\
G\phi &= \neg F\neg \phi & H\phi &= \neg P\neg \phi \\
\phi_1 R \phi_2 &= \neg (\neg \phi_1 U \neg \phi_2) & \phi_1 T \phi_2 &= \neg (\neg \phi_1 S \neg \phi_2)
\end{align*}
$$

The following example shows how behavior of a system can be formally captured employing LTL formulae.
2.4 Bounded Satisfiability Checking

System description:
The system starts with an occurrence of $ev_1$ (initialization), the occurrences of two events can not happen at the same time, and the two events occur alternatively, infinitely often (overall behavior).

Formal specification in LTL:
$ev_1 \land G((ev_1 \Rightarrow \neg ev_2) \land (ev_1 \Rightarrow X(\neg ev_1 U ev_2)) \land (ev_2 \Rightarrow X(\neg ev_2 U ev_1)))$

The LTL formula has two parts, $ev_1$ that captures the initialization of the system (the whole formula is asserted to be true at the first time instant), and the overall behavior is captured employing $G$, that is globally in the future. The formula capturing the overall behavior is comprised of three parts:

$(ev_1 \Rightarrow \neg ev_2)$ – that says if at the current time instant $ev_1$ occurs then we do not have any occurrence of $ev_2$. This can also be captured by $\neg (ev_1 \land ev_2)$ or $\neg ev_1 \lor \neg ev_2$. This formula captures “the occurrences of two events may not coincide”.

$(ev_1 \Rightarrow X(\neg ev_1 U ev_2)) \land (ev_2 \Rightarrow X(\neg ev_2 U ev_1))$ – that says having the occurrence of an event forces the occurrence of the other event and the absence of the former event until then. This formula forces the two events to occur just like the specification, that is “the two events occur alternatively, infinitely often”. They occur “alternatively” because no occurrence of $ev_1$ is allowed until next occurrence of $ev_2$ when $ev_1$ occurs, and no occurrence of $ev_2$ is allowed until next occurrence of $ev_1$ when $ev_2$ occurs. They occur “infinitely often”, because every occurrence of $ev_1$ forces an occurrence of $ev_2$ and every occurrence of $ev_2$ forces an occurrence of $ev_1$. The initialization that is an occurrence of $ev_1$ triggers this infinite behavior.

2.4 Bounded Satisfiability Checking

Bounded approaches towards satisfiability checking of LTL formula unfold the given formula up to a limited number of steps and check its satisfiability within the given bound. These approaches are the fastest to reveal system errors, but to make it complete and verify properties, a big enough $K$ must be determined by algorithms, that are expensive. However, complete approaches suffer much more from the state explosion problem and bounded approaches are preferred especially when complete ones give up in returning verification result for huge formal models. Still, scalability is a problem even in bounded approaches, that hampers verification of big formal models.

The principle that underlies BSC and also Bounded Model Checking (BMC) techniques is, given formalizations of the system $S$ and of the property $\phi$ that should hold for the system, to look for an infinite, ultimately periodic execution $\pi = s_0 s_1 \ldots s_{l-1} (s_l s_{l+1} \ldots s_k)^\omega$ of $S$ that violates $\phi$, where $k$ is a parameter (Figure 2.1). If a counterexample witnessing the violation of the property exists, then the property does not hold for $S$. If no counterexample of length up to $k$ is found, then the property holds for $S$ provided that $k$ is big enough.

In BMC the system ($S$) is modeled by a finite automaton and the property ($\phi$) is expressed in temporal logic, while in BSC, LTL is the underline formalism for modeling both the system and the property. However, both techniques can be used to check whether an LTL formula $\phi$ is satisfiable or not, provided that in BMC, no system in defined and negation of the LTL formula is introduced as the property. At the core of both BMC and BSC is the idea of translating an LTL formula $\phi$ into a formula of Propositional Logic.
Chapter 2. The Temporal Satisfiability Checking problem

that represents ultimately periodic models of $\phi$. Then, its verification is performed by feeding the translated formula to a solver.

2.5 BSC for LTL with Eventualities

In the following, we describe the classic technique for encoding LTL formulae into propositional logic introduced in [16], which is at the core of BSC. To this end, one only needs to represent states $s_0 \ldots s_l \ldots s_k$, and then the fact that the state after $s_k$, say $s_{k+1}$, is in fact $s_l$ again.

$$s_0 \ldots s_l \ldots s_k = s_l$$

**Figure 2.1:** An infinite, ultimately periodic execution with loop at position $l$.

Hence, the bounded encoding captures finite sequences of states of the form $\alpha s \beta s$, where $\alpha = s_0 s_1 \ldots s_{l-1}$, $\beta = s_{l+1} \ldots s_k$, and $s = s_l = s_{k+1}$.

The encoding is defined as Boolean constraints over so-called formula variables \([[[\psi]]]]). These are Boolean variables which are used to represent the value of all subformulae of the LTL formula to be checked for satisfiability in instants $0, 1, \ldots k + 1$. More precisely, given an LTL formula $\phi$ and a bound $k$, the encoding introduces, for each subformula $\psi$ of $\phi$, $k + 2$ formula variables \([[[\psi]]]]_0, [[[\psi]]]]_1, \ldots [[[\psi]]]]_{k+1}$ which capture whether $\psi$ is true or not at the various instants in $[0, k + 1]$.

In addition, the encoding introduces $k + 1$ loop selector variables: $l_0, l_1, \ldots, l_k$, which are fresh Boolean variables such that $l_i$ is true iff the loop starts at position $l$ (hence, if $l_i$ is true, then $s_l = s_{k+1}$); at most one of $l_0, l_1, \ldots, l_k$ can be true. Other Boolean variables are introduced for convenience: the $k + 1$ variables $InLoop_i$, with $0 \leq i \leq k$ are such that $InLoop_i$ is true iff position $i$ is in the loop (i.e., $l \leq i \leq k$). Finally, variable $LoopExists$ is true iff the desired loop exists.

In the rest of this section, we present the constraints that are imposed on the Boolean variables introduced above to capture the semantics of LTL formulae. Table 2.1 defines a set of constraints called $|LoopConstraints|_k$:

**Table 2.1:** Loop constraints.

<table>
<thead>
<tr>
<th>Base</th>
<th>$\neg l_0 \land \neg InLoop_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1 \leq i \leq k$</td>
<td>($l_i \Rightarrow s_{i-1} = s_k$) $\land$ ($InLoop_i \equiv InLoop_{i-1} \lor l_i$) $\land$ ($InLoop_{i-1} \Rightarrow \neg l_i$) $\land$ ($LoopExists \equiv InLoop_k$)</td>
</tr>
</tbody>
</table>

They essentially define the semantics of Boolean variables $\{l_i\}_{i \in [0,k]}$, $\{InLoop_i\}_{i \in [0,k]}$ and $LoopExists$ (e.g., the existence of at most one loop). In addition, as mentioned
2.5. BSC for LTL with Eventualities

In [16], they impose that the same atomic propositions that hold in state $s_k$ also hold in state $s_{l-1}$, which has been shown to improve the efficiency of the model search.

Table 2.2 shows $|LastStateConstraints|_k$.

| Base | $\neg LoopExists \Rightarrow \neg ||\phi||_{k+1}$ |
|------|--------------------------------------------------|
| $1 \leq i \leq k$ | $l_i \Rightarrow (||\phi||_{k+1} \Leftrightarrow ||\phi||_i)$ |

They define that the subformulae of $\phi$ that hold in $s_{k+1}$ are the same as those that hold in state $s_l$. This effectively defines that after state $s_k$ the bounded trace loops back to state $s_l$.

The subsequent constraints define the semantics of the propositional connectives and of the temporal operators. Table 2.3 introduces the set of constraints $|PropConstraints|_k$, which capture the semantics of propositional connectives;

<table>
<thead>
<tr>
<th>$\phi$</th>
<th>$0 \leq i \leq k + 1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p$</td>
<td>$</td>
</tr>
<tr>
<td>$\neg p$</td>
<td>$</td>
</tr>
<tr>
<td>$\psi_1 \land \psi_2$</td>
<td>$</td>
</tr>
<tr>
<td>$\psi_1 \lor \psi_2$</td>
<td>$</td>
</tr>
</tbody>
</table>

For example, $|PropConstraints|_k$ state that the value of $||p||_i$ and $||\neg p||_i$ capture whether propositional letter $p$ holds at instant $i$ or not. The definitions of $||\psi_1 \land \psi_2||_i$ and of $||\psi_1 \lor \psi_2||_i$ are straightforward. Notice that the Boolean encoding was defined for LTL formulae in Positive Normal Form (PNF), i.e., in which negations can only appear next to atomic propositions. This can save some formula variables, but the encoding can be easily generalized to formulae that are not in PNF.

The rest of this chapter defines the semantics of the temporal operators, both future ($X$, $U$ and $R$) and past ones ($Y$, $S$ and $T$). We call this whole set of constraints $|TempConstraints|_k$.

The semantics of $U$ and $R$ is defined through their standard fixpoint characterization:

The definition of the semantics of $U$ and $R$ is completed through the introduction of the set of constraints $|Eventualities|_k$, which are presented in the Table 2.5.

These constraints are used to make sure that, if $\psi_1 U \psi_2$ holds in $s_k$, then $\psi_2$ occurs infinitely often, that is, it occurs somewhere in the loop. Similarly, if $\psi_1 R \psi_2$ occurs in $s_k$, then either $\psi_2$ holds throughout the loop, or at some point of the loop $\psi_1$ holds. $\langle\langle F \psi_2 \rangle\rangle_i$ and $\langle\langle G \psi_2 \rangle\rangle_i$, are auxiliary variables required for capturing these constraints. $\langle\langle F \psi_2 \rangle\rangle_i$ holds if position $i$ belongs to the loop and $\psi_2$ holds in at least one position.
between \( l \) and \( i \). Accordingly, \( \langle \langle F \psi_2 \rangle \rangle_k \) means that \( \psi_2 \) holds somewhere in the loop. Therefore, constraint \( \text{LoopExists} \Rightarrow (\langle [\psi_1 \text{U} \psi_2] \rangle_k \Rightarrow \langle \langle F \psi_2 \rangle \rangle_k) \) does not allow \( \psi_1 \text{U} \psi_2 \) to hold at \( k \), if \( \psi_2 \) does not occur infinitely often. Similarly, \( \langle \langle G \psi_2 \rangle \rangle_k \) holds iff \( \psi_2 \) holds everywhere in the loop. Then, constraint \( \text{LoopExists} \Rightarrow (\langle [\psi_1 \text{R} \psi_2] \rangle_k \Leftarrow \langle \langle G \psi_2 \rangle \rangle_k) \) forces \( \langle [\psi_1 \text{R} \psi_2] \rangle_k \) to hold if \( \psi_2 \) holds from position \( l \) on.

Table 2.6 defines the semantics of the past operators \( Y, S \) and \( T \), which is symmetrical to their future counterparts.

### Table 2.6: Past temporal subformulae constraints.

<table>
<thead>
<tr>
<th>( \phi )</th>
<th>( 0 &lt; i \leq k + 1 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( Y \psi )</td>
<td>([Y\psi]<em>i \Leftarrow [\psi]</em>{i-1})</td>
</tr>
<tr>
<td>( Z \psi )</td>
<td>([Z\psi]<em>i \Leftarrow [\psi]</em>{i-1})</td>
</tr>
<tr>
<td>( \psi_1 S \psi_2 )</td>
<td>([\psi_1 S \psi_2]_i \Leftarrow [\psi_2]_i \vee ([\psi_1]<em>i \wedge [\psi_1 S \psi_2]</em>{i-1}))</td>
</tr>
<tr>
<td>( \psi_1 T \psi_2 )</td>
<td>([\psi_1 T \psi_2]_i \Leftarrow [\psi_2]_i \wedge ([\psi_1]<em>i \vee [\psi_1 T \psi_2]</em>{i-1}))</td>
</tr>
</tbody>
</table>

It also defines operator \( Z \), which is necessary for formulae in PNF, which is simply a variant of \( Y \) such that \( Z \psi \) holds in \( 0 \) no matter \( \psi \). Since the temporal domain is mono-infinite (i.e., it is infinite only towards the future), there is no need to impose eventuality constraints over past operators. However, we must define the value of past...
2.6. Constraint LTL (CLTL) and CLTL Over Clocks (CLTLoc)

operators in the origin 0. This is done through the constraints of Table 2.7.

Table 2.7: First state constraints.

<table>
<thead>
<tr>
<th>$\phi$</th>
<th>Base</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Y\psi$</td>
<td>$\neg[[Y\psi]]_0$</td>
</tr>
<tr>
<td>$Z\psi$</td>
<td>$[[Z\psi]]_0$</td>
</tr>
<tr>
<td>$\psi_1 S \psi_2$</td>
<td>$[[\psi_1 S \psi_2]]_0 \iff [[\psi_2]]_0$</td>
</tr>
<tr>
<td>$\psi_1 T \psi_2$</td>
<td>$[[\psi_1 T \psi_2]]_0 \iff [[\psi_2]]_0$</td>
</tr>
</tbody>
</table>

Finally, given an LTL formula $\phi$, its Boolean encoding $\phi_B$ is given by the conjunction of the constraints in sets $|LoopConstraints|_k$, $|LastStateConstraints|_k$, $|PropConstraints|_k$, $|TempConstraints|_k$, and $|Eventualities|_k$, plus the statement that $\phi$ holds in the origin, i.e. $[[\phi]]_0$.

2.6 Constraint LTL (CLTL) and CLTL Over Clocks (CLTLoc)

Two of the most important factors that still hamper the applicability of LTL-based approaches in practice are the limited efficiency and scalability of the corresponding verification tools, and the lack of expressiveness of the logic, which does not allow one to express, for example, variables with infinite domains (unbounded integers or reals). The bit-vector-based encoding presented Chapter 5 addresses the first issue. To address the second concern, one can extend the basic, propositional version of LTL with first-order constructs, and in particular with infinite-domain variables, while still keeping the resulting logic decidable, hence verifiable in a fully automated way.

Let us introduce a first motivating example for the need for infinite domains in LTL. Consider the classic leader election protocol introduced in [27]. The goal of the protocol is to elect a leader in a ring of processes that exchange messages. Each process in the ring chooses a number, and communicates this number to its immediate neighbor on one side. The processes then engage in a sequence of actions (receiving and sending numbers, comparing received numbers with stored ones) until the leader is elected. The selected leader is the one that had initially chosen the highest number among those in the ring. The protocol is guaranteed to elect a unique leader when the initially chosen numbers are all distinct. The protocol is rather simple, and it can be formally verified, for example, through the Spin model checker\(^3\). However, the Promela\(^4\) model analyzed by Spin can only deal with finite ranges of integer values, so all we can verify through the tool is that, if the numbers assigned to processes are taken from a certain finite domain, the protocol will work correctly.

Constraint LTL (CLTL) [30] is a first-order extension of LTL that allows variables to take values from infinite domains such as integers or reals; the values assigned to variables at a time instant can be compared against each other (i.e., one can write

\(^3\)The protocol is one of the basic examples included in the tool distribution [77].

\(^4\)Promela is the input language of the Spin model checker.
Chapter 2. The Temporal Satisfiability Checking problem

constraints such as \( x < y \), and against their future values (e.g., the value of a variable in the next instant). Through CLTL it is possible to create a model of the leader election protocol that allows for numbers assigned to processes in the ring to take values in an infinite domain such as the set of integers.

When variables can be real-valued, it becomes natural to introduce the possibility that they behave as clocks that measure the passing of time, as in Timed Automata [2]; that is, the value of each variable (i.e., clock) does not change through assignment, but it increases with the passing of time. In fact, real-valued clocks are a typical mechanism for quantitatively modeling the passing of time in systems that combine both software components and physical ones, such as embedded and cyber-physical systems. LTL-like temporal logics that include a notion similar to that of time-measuring clocks have been studied for several decades. Timed Propositional Temporal Logic (TPTL) [3] is a classic example of such logics, but it becomes undecidable when clocks are real-valued. Then, as a second motivating example, we would like to have a decidable, real-time, extension of LTL that allows us to capture the passing of time in a quantitative way, where time quantities are real-valued.

In the following, we first introduce the CLTL logic and its extension where variables behave as clocks (CLTL-over-clocks).

**Constraint LTL** (CLTL [10, 30]) is a decidable fragment of First-Order LTL. CLTL formulae are defined with respect to a finite set \( V \) of variables and a constraint system \( D \), which is a pair \((D, \mathcal{R})\) with \( D \) being a specific domain of interpretation for variables and constants and \( \mathcal{R} \) being a family of relations on \( D \), such that the set \( \mathcal{AP} \) of atomic propositions coincides with set \( \mathcal{R}_0 \) of 0-ary relations. An atomic constraint is a term of the form \( R(x_1, \ldots, x_n) \), where \( R \) is an \( n \)-ary relation on \( D \) and \( x_1, \ldots, x_n \) are variables. A valuation is a mapping \( v : V \rightarrow D \), i.e., an assignment of a value in \( D \) to each variable. A constraint is satisfied by \( v \), written \( v \models_D R(x_1, \ldots, x_n) \), if \((v(x_1), \ldots, v(x_n)) \in R \). Given a variable \( x \in V \) over domain \( D \), temporal terms are defined by the syntax: \( \alpha := c \mid x \mid X\alpha \mid Y\alpha \mid \phi U \phi \mid \phi S \phi \)

where \( \alpha_i \)'s are temporal terms, \( R \in \mathcal{R}, X, Y, U \) and \( S \) are the usual “next”, “previous”, “until” and “since” operators of LTL, with the same meaning. Operators “eventually” \( F \), and “globally” \( G \) are defined as usual, i.e., \( F\psi \) is \( \top U \psi \) and \( G\psi \) is \( \neg F(\neg \psi) \).

The semantics of CLTL formulae is defined with respect to a strict linear order representing time \((\mathbb{N}, <)\). The truth values of propositions in \( \mathcal{AP} \), and values of variables belonging to \( V \) are defined by a pair \((\pi, \sigma)\) where \( \sigma : \mathbb{N} \times V \rightarrow D \) is a function that defines the value of variables at each position in \( \mathbb{N} \) and \( \pi : \mathbb{N} \rightarrow \mathcal{AP}(\mathcal{AP}) \) is a function associating a subset of the set of propositions with each element of \( \mathbb{N} \). The value of terms is defined with respect to \( \sigma \) as follows:

\[
\sigma([x]) = c
\]

where \( x_\alpha \) is the variable in \( \alpha \) occurring in term \( \alpha \), and \( |\alpha| \) is the depth of a temporal term, namely the total amount of temporal shift needed in evaluating \( \alpha \): \( |x| = 0 \) when \( x \).
is a variable, and $|X\alpha| = |\alpha| + 1$. The semantics of a CLTL formula $\phi$ at instant $i \geq 0$ over a linear structure $(\pi, \sigma)$ is recursively defined as follows.

$$(\pi, \sigma), i \models p \iff p \in AP$$

$$(\pi, \sigma), i \models R(\alpha_1, \ldots, \alpha_n) \iff (\sigma(i + |\alpha_1|, x_{\alpha_1}), \ldots, \sigma(i + |\alpha_n|, x_{\alpha_n})) \in R$$

$$(\pi, \sigma), i \models -\phi \iff (\pi, \sigma), i \not\models \phi$$

$$(\pi, \sigma), i \models \phi \land \psi \iff (\pi, \sigma), i \models \phi \land (\pi, \sigma), i \models \psi$$

$$(\pi, \sigma), i \models X\phi \iff (\pi, \sigma), i + 1 \models \phi$$

$$(\pi, \sigma), i \models Y\phi \iff (\pi, \sigma), i - 1 \models \phi \land i > 0$$

$$(\pi, \sigma), i \models \phi U \psi \iff \exists j \geq i: (\pi, \sigma), j \models \psi \land (\pi, \sigma), n \models \phi \forall i \leq n < j$$

$$(\pi, \sigma), i \models \phi S \psi \iff \exists 0 \leq j \leq i: (\pi, \sigma), j \models \psi \land (\pi, \sigma), n \models \phi \forall j < n \leq i$$

A formula $\phi \in$ CLTL is satisfiable if there exists a pair $(\pi, \sigma)$ such that $(\pi, \sigma), 0 \models \phi$.

We are particularly interested in the cases where $D = (\mathbb{Z}, \{<, =\})$ and $D = (\mathbb{R}, \{< , =\})$, which are known to be decidable [30], and a decision procedure based on bounded satisfiability checking mechanisms has been defined in [10]. This decision procedure has been implemented in the ae2zot plugin of the Zot tool [64]. To illustrate the features of the language, the next CLTL formula states that, each time predicate swap_a_and_b holds, the values of variables $a$ and $b$ are swapped, that is, the next value of variable $a$ is equal to the current value of variable $b$, and vice-versa:

$$G(swap_a_and_b \Rightarrow (Xa = b \land Xb = a)) . \tag{2.1}$$

CLTL-over-clocks (CLTLoc) is a variant of CLTL, where $D = (\mathbb{R}, \{<, =\})$, arithmetic variables are evaluated as clocks, and the arithmetic “next” operator $X$ is not allowed. A clock “measures” the time elapsed since the last time the clock was “reset” (i.e., the variable was equal to zero). By definition, in CLTLoc each $i \in \mathbb{N}$ is associated with a “time delay” $\delta(i)$, where $\delta(i) > 0$ for all $i$, which corresponds to the “time elapsed” between $i$ and the next state $i + 1$. More precisely, for all clocks $x \in V$, $\sigma(i + 1, x) = \sigma(i, x) + \delta(i)$, unless it is “reset” (i.e., $\sigma(i + 1, x) = 0$). It is shown in [14] that CLTLoc is decidable. In addition, [13] shows that CLTLoc is equivalent to Timed Automata, so it is well suited for capturing timed specifications.

For example, the following CLTLoc formula states that, when predicate turn_on holds, a clock $x$ is reset (i.e., it is equal to 0), and then predicate on holds until $x$ hits value 5 (i.e., the light stays on for at least 5 time units):

$$G(turn_on \Rightarrow (x = 0 \land X(x > 0 \land on)U(x = 5 \land on))) . \tag{2.2}$$

2.7 TRIO Metric Temporal Logic

TRIO [22] is an temporal logic that supports a quantitative metric on time. TRIO formulae constructs are propositional connectives, quantifiers, and a basic modal operator, namely Dist, that refers to the truth value of a subformula $\phi$ at distance $t$ from the current time instant. For example, (Dist($\phi$, 0)), (Dist($\phi$, $t$)), and (Dist($\phi$, $-t$)) with $t > 0$ refer to the value of subformula $\phi$ at the current time instant, at $t$ time instants in the future, and at $t$ time instants in the past, respectively. While TRIO can exploit both discrete and dense time domains, in the problem being investigated in this thesis the time domain is
Table 2.8: TRIO temporal operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Futr(φ, t)</td>
<td>( t \geq 0 \land \text{Dist}(\phi, t) )</td>
</tr>
<tr>
<td>Past(φ, t)</td>
<td>( t \geq 0 \land \text{Dist}(\phi, -t) )</td>
</tr>
<tr>
<td>Alw(φ)</td>
<td>( \forall d : \text{Dist}(\phi, d) )</td>
</tr>
<tr>
<td>AlwF(φ)</td>
<td>( \forall d \geq 0 : \text{Futr}(\phi, d) )</td>
</tr>
<tr>
<td>AlwP(φ)</td>
<td>( \forall d \geq 0 : \text{Past}(\phi, d) )</td>
</tr>
<tr>
<td>Somφ</td>
<td>( \exists d : \text{Dist}(\phi, d) )</td>
</tr>
<tr>
<td>SomF(φ)</td>
<td>( \exists d \geq 0 : \text{Futr}(\phi, d) )</td>
</tr>
<tr>
<td>SomP(φ)</td>
<td>( \exists d \geq 0 : \text{Past}(\phi, d) )</td>
</tr>
<tr>
<td>Lasts(φ, t)</td>
<td>( \forall d \in (0, t] : \text{Futr}(\phi, d) )</td>
</tr>
<tr>
<td>Lasted(φ, t)</td>
<td>( \forall d \in (0, t] : \text{Past}(\phi, d) )</td>
</tr>
<tr>
<td>WithinP(φ, t)</td>
<td>( \exists d \in (0, t] : \text{Past}(\phi, d) )</td>
</tr>
<tr>
<td>WithinF(φ, t)</td>
<td>( \exists d \in (0, t] : \text{Futr}(\phi, d) )</td>
</tr>
<tr>
<td>Since(ψ, φ)</td>
<td>( \exists d \geq 0 : \text{Lasted}(\psi, d) \land \text{Past}(\phi, d) )</td>
</tr>
<tr>
<td>Until(ψ, φ)</td>
<td>( \exists d \geq 0 : \text{Lasts}(\psi, d) \land \text{Futr}(\phi, d) )</td>
</tr>
</tbody>
</table>

discrete, mono-infinite, left-closed, right-open. In other words, the time domain is \( \mathbb{N} \) and time instants are non-negative integers.

From the basic operators various operators can be defined as abbreviations for the sake of convenience and readability of specification formulae. These abbreviations are captured in Table 2.8.

For the same reason, different variants of temporal operators based on inclusion and exclusion of interval endpoints are introduced (Table 2.9). These operators follow the format of \( \text{Operator}_{ab} \), such that \( a, b \in \{i, e\} \), and \( i \) and \( e \) stand for included and excluded respectively. For the future operators like Until and Lasts, \( a \) and \( b \) refers to the left and right end points, respectively, while for the past temporal operators like Since and Lasted, \( a \) and \( b \) refers to the right and left end points, respectively.

In addition, in TRIO formulae any free variable is implicitly universally quantified at the outermost level, and the whole formula is implicitly temporally closed with the Alw (always) operator. A set of basic items that are predicates, time-dependent variables and functions build the TRIO specification that represents the overall state of the system. The behavior of the system is defined by capturing constraints over these predicates.

**TRIO to LTL/CLTL.**

Although TRIO over discrete-time domain does not add any expressiveness power to LTL/CLTL, it is more suitable for formal specification of complex systems owing to its conciseness. Discrete-time TRIO formulae can be translated to CLTL/LTL, provided that we take each step as one time instant. Suppose that we want to formalize “If the temperature has been less than 20 centigrade, seamlessly for 5 time instants, then ev1 must occur.” In CLTL it is formulated as:

\[
(\text{Y}(\text{temp} < 20) \land \\
\text{Y(Y(temp < 20)))} \land \\
\text{Y(Y(Y(temp < 20))))} \land
\]
2.7. TRIO Metric Temporal Logic

<table>
<thead>
<tr>
<th>Operator</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{Lasts}_{\text{cs}}(\phi, t)$</td>
<td>$\forall d \in (0, t] : \text{Futr} (\phi, d)$</td>
</tr>
<tr>
<td>$\text{Lasts}_{\text{sc}}(\phi, t)$</td>
<td>$\forall d \in [0, t] : \text{Futr} (\phi, d)$</td>
</tr>
<tr>
<td>$\text{Lasts}_{\text{si}}(\phi, t)$</td>
<td>$\forall d \in [0, t] : \text{Futr} (\phi, d)$</td>
</tr>
<tr>
<td>$\text{Lasts}_{\text{ce}}(\phi, t)$</td>
<td>$\forall d \in (0, t] : \text{Futr} (\phi, d)$</td>
</tr>
<tr>
<td>$\text{Until}_{\text{le}}(\psi, \phi)$</td>
<td>$\exists d \geq 0 : \text{Lasts}_{\text{le}} (\psi, d) \land \text{Futr} (\phi, d)$</td>
</tr>
<tr>
<td>$\text{Until}_{\text{li}}(\psi, \phi)$</td>
<td>$\exists d \geq 0 : \text{Lasts}_{\text{li}} (\psi, d) \land \text{Futr} (\phi, d)$</td>
</tr>
<tr>
<td>$\text{Until}_{\text{ei}}(\psi, \phi)$</td>
<td>$\exists d \geq 0 : \text{Lasts}_{\text{ei}} (\psi, d) \land \text{Futr} (\phi, d)$</td>
</tr>
<tr>
<td>$\text{Since}_{\text{le}}(\psi, \phi)$</td>
<td>$\forall d \in (0, t] : \text{Past} (\phi, d)$</td>
</tr>
<tr>
<td>$\text{Since}_{\text{li}}(\psi, \phi)$</td>
<td>$\forall d \in [0, t] : \text{Past} (\phi, d)$</td>
</tr>
<tr>
<td>$\text{Since}_{\text{ei}}(\psi, \phi)$</td>
<td>$\forall d \in [0, t] : \text{Past} (\phi, d)$</td>
</tr>
<tr>
<td>$\text{Since}_{\text{ce}}(\psi, \phi)$</td>
<td>$\forall d \in (0, t] : \text{Past} (\phi, d)$</td>
</tr>
<tr>
<td>$\text{Since}_{\text{ei}}(\psi, \phi)$</td>
<td>$\exists d \geq 0 : \text{Lasts}_{\text{ei}} (\psi, d) \land \text{Past} (\phi, d)$</td>
</tr>
<tr>
<td>$\text{Since}_{\text{le}}(\psi, \phi)$</td>
<td>$\exists d \geq 0 : \text{Lasts}_{\text{le}} (\psi, d) \land \text{Past} (\phi, d)$</td>
</tr>
<tr>
<td>$\text{Since}_{\text{ei}}(\psi, \phi)$</td>
<td>$\exists d \geq 0 : \text{Lasts}_{\text{ei}} (\psi, d) \land \text{Past} (\phi, d)$</td>
</tr>
<tr>
<td>$\text{Since}_{\text{ce}}(\psi, \phi)$</td>
<td>$\exists d \geq 0 : \text{Lasts}_{\text{ce}} (\psi, d) \land \text{Past} (\phi, d)$</td>
</tr>
</tbody>
</table>

Table 2.9: Variants of TRIO temporal operators

\[
\text{Y}(\text{Y}(\text{Y}(\text{temp} < 20)))) \land \\
\text{Y}(\text{Y}(\text{Y}(\text{Y}(\text{temp} < 20)))) \Rightarrow \text{ev}_1
\]

whereas in TRIO it is simply expressed as:

Lasted(\text{temp} < 20, 5) \Rightarrow \text{ev}_1.

The translation of TRIO formulae into LTL/CLTL formulae is straight forward. 
$\text{Futr}(\phi, 1)$ and $\text{Futr}(\phi, 2)$ are equivalent to $\text{X}(\phi)$ and $\text{X}(\text{X}(\phi))$ respectively, and there are as many nested $\text{X}$ operators as the value of the second parameter of $\text{Futr}$. $\text{Lasts}(\phi, 2)$ and $\text{WithinF}(\phi, 2)$ are translated as $\text{X}(\phi) \land \text{X}(\text{X}(\phi))$ and $\text{X}(\phi) \lor \text{X}(\text{X}(\phi))$, respectively. More precisely the future temporal operators are translated as the following:

\[
\text{Futr}(\phi, i) = \begin{cases} 
\text{X}(\phi), & \text{if } \ i = 1 \\
\text{X}(\text{Futr}(\phi, i - 1)), & \text{otherwise}
\end{cases}
\]

\[
\text{Lasts}(\phi, i) = \begin{cases} 
\text{X}(\phi), & \text{if } \ i = 1 \\
\text{X}(\phi \land \text{Lasts}(\phi, i - 1)), & \text{otherwise}
\end{cases}
\]

\[
\text{WithinF}(\phi, i) = \begin{cases} 
\text{X}(\phi), & \text{if } \ i = 1 \\
\text{X}(\phi \lor \text{WithinF}(\phi, i - 1)), & \text{otherwise}
\end{cases}
\]

The past counterparts of $\text{Futr}$, $\text{Lasts}$, and $\text{WithinF}$ – $\text{Past}$, $\text{Lasted}$, and $\text{WithinP}$, respectively – are translated like them, where $\text{X}$ is replaced by $\text{Y}$.

Finally, $\text{Alw}$ and $\text{Som}$ are the only operators that predicate over both future and past instance, and are translated as follows:

\[
\text{Alw}(\phi) = \text{G}(\phi) \land \text{H}(\phi)
\]
Chapter 2. The Temporal Satisfiability Checking problem

where $G$ and $H$ are “globally” and “historically” in CLTL/LTL, respectively.

$$\text{Som}(\phi) = F(\phi) \lor P(\phi)$$

where $F$ and $P$ are “eventually in the future” and “eventually in the past” in CLTL/LTL, respectively.

2.8 Related Work

There are essentially two groups of approaches to the problem of satisfiability checking of LTL formulae: bounded and automata-based. In automata-based approaches the LTL formula is translate to a Büchi automaton that is a finite automata on infinite words. The given LTL formula is satisfiable if there exist a run of the corresponding automaton that visit one or more final states infinitely often and it is unsatisfiable otherwise.

Rozier and Vardi [73] carried out a comparison of satisfiability checkers for LTL formulae based on the translation of LTL formulae into Büchi automata. Rozier and Vardi [74] also propose a novel translation of LTL formulae into Transition-based Generalized Büchi Automata, inspired by the translation presented in [41]. Such automata are used by SPOT [32], which is claimed to be the best explicit LTL-to-Büchi automata translator for satisfiability checking purposes based on the experiments carried out in [73]. Li et al. [58] present a novel on-the-fly construction of Büchi automata from LTL formulae that is particularly well suited for finding models of LTL formulae when they exist. The advantage of automata-based approaches is that they are complete and unlike bounded approaches no further step is required. On the other hand, they suffer much more from the state explosion problem.

Given the different nature of our approaches with respect to automata-based ones, however, we do not compare our approaches against them. This thesis focuses on similar, BSC-based approaches that are compared with one another in Chapter 6. Compared approaches includes LTL encoding with eventuality and similar ones presented in [16], [56], [57], and [49], which are implemented in the well-known NuSMV model checker [23].

A simple translation of LTL formula to CNF (Conjunctive Normal Form) formula is presented in [56], regarding the semantic equivalence of LTL and Computation Tree Logic (CTL), when each step has only one successor in the Kripke structure. Another bounded encoding is presented in [57] that virtually unroll the path up to the number of maximum past operator depth ($d$) in the LTL formula. Unlike other approaches, in this encoding the LTL formula is unfolded up to $d \times k$ steps, instead of $k$.

As for CLTL encodings, recently, a decision procedure based on a bounded satisfiability approach for the automated verification of CLTL specifications is presented in [10]. To the best of our knowledge, this decision procedure is the first one to be implemented for CLTL; it is available as part of our Zot tool [64].

Finally, an exhaustive evaluation of several techniques and tools (including some that are not based on translation to Büchi automata or on bounded approaches) can be found in [75]. Still, scalability is a problem in all the aforementioned approaches even in bounded ones,] that hampers verification of big formal models.
UML is a general-purpose, platform independent modeling language. As an industrial standard, it is frequently used for design and analysis of software systems. Aligned with MDE, it enables designing of a system independently of the implementation choices, such that the model can be realized on different platforms. However, it has been often criticized [31, 34] because of the too many diagram types and symbols and because of the complexity of managing structural and behavioral views spread over many different diagrams. The semantics provided by the OMG [67] is given through textual descriptions and it is often ambiguous. UML specification consists of a collection of loosely coupled models (classes, use cases, collaborations, activities, etc.) that are tied together by few and semantically weak rules [42]. The introduction of domain-specific extensions (profiles, according to the OMG group) further complicates the integration of standard and specific concepts.

These problems have often hampered the use of UML. There have been many different attempts to ascribe the notation with a (more) formal semantics, but the wideness of the language has often led the authors to only concentrate on some diagram types (e.g., state and collaboration diagrams), while neglecting the key characteristics of UML, that is the rich set of diagram types and the freedom with which the designer can model a system.

Since this work is a continuation of Alfredo Motta’s PhD dissertation [63], whose outcome is eFT-UML (extensible Formal, Timed UML), in this chapter the structure of this coherent framework and its TRIO-based formalization will be presented.

The formalization of single diagram types has been carried out several times [19, 35, 78], and the interdependencies among different diagrams were neglected, whereas the integration in a single coherent framework of a significant number of diagram types is a distinguished characteristic of eFT-UML. It proposes a comprehensive framework that accommodates Class Diagram (CD), Object Diagram (OD), Interaction Overview
Chapter 3. eFT-UML

Diagram (IOD), State Machine (SM), and Sequence Diagram (SD) interacting with one another by means of a precise set of shared events. In addition, it borrows from MARTE [66] (the UML Profile for Modeling and Analysis of Real-Time and Embedded Systems) the notion of time. This means that eFT-UML is particularly suited for the specification of timed systems, that are systems whose correctness is not just functional, but it also depends on the delays associated with the different activities and with the temporal ordering of the different events. eFT-UML also extends UML with Time Property Diagrams (TPDs) to let the user render the LTL (or TRIO) properties of interest through the same graphical elements. The formal semantics of eFT-UML is based on the TRIO [22] metric temporal logic, which gives us the flexibility and composability required to specify the semantics of a complex notation.

The chapter is organized as follows. Section 3.1 presents the structure of eFT-UML models and gives an overview of valid eFT-UML models. The dimension of time, a special feature of eFT-UML, is presented in Section 3.2. The formal semantics of different diagram types are presented in Section 3.3. Finally, Section 3.4 surveys the works related to UML formalization.

3.1 Structure

A valid eFT-UML model must comprise at least a CD, while all the behavioral views (like SM and SD) are optional. CDs provide the static definitions of the elements in a system. Every class should have at least one Instance Specification (object) instantiating the class. The number of Instance Specifications must be finite in order to run the formal verification. CDs can also introduce Clock Types, then used in the system to constrain the timed behavior of components [4, 67].

![Diagram showing the structure of a valid eFT-UML model.](image)

Figure 3.1: Structure of valid eFT-UML model.

SMs can be used to describe the behavior of significant classes in the system; that is, all their instances will behave as stated by these diagrams. However, each class may be associated with one or more SMs, which would then run in parallel. A single object will then be into multiple states at a given time. Accordingly, each of these states belongs to
one of the SMs assigned to that object.

SDs describe partial behaviors (scenarios) of the system. They identify the messages exchanged between the instances of the classes defined in CDs. These messages in turn should be instances of the operations defined in the class the object belongs to. IODs constitute high-level structuring mechanism that is used to compose SDs through standard operators such as sequence, iteration, concurrency, and choice [5].

3.2 Time Dimension

In addition to standard UML diagrams, the notation of time constraint is added to predicate on the time dimension of system. Special purpose comments, tagged with stereotype ≪TimeConstraint≫ can be used to annotate the temporal distance between occurrence of two events. They always have the following format:

\[ ev2 - ev1 \sim K \]

Where \( ev1 \) and \( ev2 \) are the events occurring in the model, \( ev2 - ev1 \) is the number of time instants elapsed since \( ev1 \) occurred until the occurrence of \( ev2 \) (i.e., the difference of their timestamps), \( \sim \) is a relation in the set \{<, ≤, ==, !=, ≥, >\}, and \( K \) is a numeric constant (integer).

eFT-UML addresses the interdependencies among the different diagrams of a complete specification by means of shared events. These events include operation invocations, initiation, completion, and suspension of SDs, initiation and completion of IODs, clock ticks, and signals defined in CDs. More specifically, inspired by MARTE [29], a notation is defined to refer to an event: \(@id.extension\) where \( id \), the name of corresponding element, and \( extension \) can be:

- \(@ID.start, @ID.end, @ID.stop\), where \( ID \) is the name of an SD and the three events refer to the time instants at which the sequence diagram starts, ends (terminates), and is stopped (interrupted). An SD ends when it accomplishes a successful scenario. If the interactions are not completed because of an interruption, we say that SD is stopped.
- \(@ID.start, @ID.end\), where \( ID \) is the name of a message in an SD, and the two events refer to the time instants at which the message is sent and received.
- \(@ID.enter, @ID.exit\), where \( ID \) is the name of a state in an SM and the two events refer to the time instants at which the corresponding object enters and exits the state.
- \(@ID.call\), where \( ID \) is the name of an operation belonging to a class in a CD and the occurrence of \( call \) signifies that the operation is invoked. For example, given an operation \( op \) defined in a CD, when an SD sends a message to that operation, event \( @op.call \) occurs.
- \(@ID.tick\), where \( ID \) is the name of a clock instance and the \( tick \) event occurs periodically and the length of period is defined in the corresponding CD.
- \(@ID.sig\), where \( ID \) is the name of a signal defined in a CD and it may be triggered (\( sig \)) because (i) it is associated with an action of a transition
in an SM, (ii) it is associated with a send signal action in an interaction overview diagram, or (iii) nondeterministically when any of the previous options have been used in the model, i.e. the signal is left free.

- \texttt{@ID.iodstart.ID.iodend: ID} is the name of an interaction overview diagram of the model and the two events refer to the time instants at which it starts and ends its execution.

- \texttt{@now} is a special event and refers to the current time instant.

As an example, if \texttt{mystate} is the name of a state in an SM, \texttt{@mystate.enter} and \texttt{@mystate.exit} refer, respectively, to the entering and exiting of state \texttt{mystate}, and \texttt{@mystate.exit} - \texttt{@mystate.enter} is the temporal distance between the two events – that is, the duration of object staying in \texttt{mystate}.

TPDs exploit class hierarchies to express TRIO metric temporal logic formulae using CDs. Each element that can appear in a TRIO formula has a corresponding class that can be used in TPDs. Then, a TPD can include: (i) standard Boolean operators (classes \texttt{Or}, \texttt{And} or \texttt{Not}, etc.); (ii) temporal operators (will be introduced in next the section), captured by classes such as \texttt{Futr}, \texttt{Alw}, \texttt{Lasts}; (iii) atomic elements, captured by classes \texttt{Term}, \texttt{BooleanExpression} and \texttt{Constant}. More precisely, class \texttt{Term} is used to refer to items such as a message in an SD or a state in an SM. Class \texttt{BooleanExpression} is used to define time constraints and inequalities over the variables declared in the UML model. Finally, \texttt{Constant} is used to introduce values — e.g., the duration of time intervals in TRIO operators.

\texttt{eFT-UML} diagrams provide users with a comprehensive, homogeneous set of concepts. More aspects, for example borrowed from MARTE, could have been added to \texttt{eFT-UML}; nevertheless, the focus is on a limited set of elements to offer designers a coherent notation that allows even non-experts to carry out formal verification activities.

### 3.3 Syntax and Semantics

The semantics of \texttt{eFT-UML} is defined through the TRIO metric temporal logic [22] and all diagrams are translated into a set of metric temporal logic predicates/variables and a set of axioms. The predicates and variables are bidirectionally mapped to the elements in diagrams, to enable the axioms ascribe their behavior in the formal model exactly as it is in the original model, and also translate back the verification result.

More formally, let us define \( D = \{CD \cup IOD \cup SM \cup SD\} \) as the set of diagrams included in an \texttt{eFT-UML} model, \( P \) as the set of predicates/variables encoding them, and \( A_P \) as the set of axioms constraining the behavior of the elements in \( P \). The \texttt{eFT-UML} semantics is a triple \( \langle \Gamma, \Delta, \Theta \rangle \), where \( \Gamma : D \rightarrow 2^P \) is a function that given a diagram returns the set of predicates/variables assigned to it, \( \Delta : D \rightarrow 2^{A_P} \) is a function that given a diagram returns the set of axioms that constrain its behavior, and \( \Theta : D \times D \rightarrow 2^{A_P} \) is a function that given two diagrams returns the set of axioms that ascribe their combined behavior. The metric temporal logic semantics \( Sem \) of an \texttt{eFT-UML} model is built as follows:

\[
Sem = \left( \bigcup_{d \in D} \Delta(d) \right) \cup \left( \bigcup_{d_1, d_2 \in D, d_1 \neq d_2} \Theta(d_1, d_2) \right)
\]
where the first part of the formula includes all the axioms that state the behavior of the different views in isolation. The second part of the formula includes all the axioms that specify the communication between them. This choice enables a formalization/verification approach that emphasizes:

- **Decoupling.** The semantics is decoupled from the predicates that represent the elements of eFT-UML models. This means that one can change the semantics while keeping the translation from UML to the predicates unchanged. For example, if we wanted to change the semantics of time associated with clocks, we do not need to alter predicates, but only the axioms associated with them. This also gives the opportunity to experiment and evaluate different semantics for the same model.

- **Extensibility.** The logic-based formalization is easy to extend. Adding a new diagram type entails defining the predicates that represent its elements, and their associated axioms. If the new diagram type shares some predicates with the already existing ones, then the coordination between diagrams is obtained seamlessly through the formulae that predicate on shared elements.

- **Coverage flexibility.** Introducing new details may result in specifications that become too big to be analyzed automatically. In such scenarios it is common practice to select an interesting subset of the model that focuses on a particular feature to simplify the analysis. To this end, analysis of partial models is supported by simply avoiding the translation of the diagrams the user is not interested in.

The metric temporal logic formalization provides a simple and straightforward mechanism to ascribe a semantics to shared events by simply adding the axioms defining the combined behavior of the different diagrams. The set of axioms returned by \( \Theta : D \times D \rightarrow 2^{A_P} \) formalizes the relation among the different diagrams by constraining the combined behavior of their predicates.

The following sections briefly present some of diagram types and their semantics, that is the output of functions \( \Gamma \) and \( \Delta \). Here, we only introduce CDs, SMs, and SDs, in order to describe how different diagrams can interact with one another via shared events and how the overall behavior of the prospective system is captured by a single TRIO formula.

### 3.3.1 Class Diagram

An eFT-UML Class Diagram CD includes classes, a finite set of objects that instantiate them, associations, clocks, and signals. A set of objects that share the same features, constraints, and semantics is described by a class whose features are attributes and operations [67]. Figure 3.2 shows an example of class notation, that contains three parts. The first part from top demonstrates the name of the class. It is written in bold and center, while the first letter is in uppercase. The attributes of the class go to the second part, while they are printed left-aligned and the first letter is in lowercase. The third part contains the operations that this class owns and can execute. The operations are printed left-aligned and the first letter is in lowercase. The visibility of attributes and operations may be included before the name of the element using a visibility symbol (+ for public, - for private, and # for protected). If no visibility symbol is introduced, it means that the attribute or the operation is public.
An association defines a semantic relationship between two or more typed instances. A link is an instance of an association. Multiplicity can be added to the association ends, to indicate how many objects from the connected class can be engaged in the relation defined by the association. Multiplicity can be: exactly N in all cases (N), a value between N and M (N..M), more than zero (+), or any value (*). If the multiplicity is not shown on an association end, it means that the multiplicity is exactly 1.

The set of objects must be finite to enable the verification. As shown in Figure 3.3 each class/object may have a clock defining periodic events (clock ticks), while signals represent general purpose events that are not bound to a specific class. The notion of clock in eFT-UML is an abstraction view of time, that is required to capture periodic events/behaviors in the specification of timed systems. For every clock \( c \), a temporal logic predicate \( \text{Clock}_{id_c} \text{Tick} \) is declared, where \( id_c \) is the unique UML identifier of the clock. Each signal \( s \) has a temporal logic predicate \( \text{Signal}_{id_s} \). For every operation \( y \) belonging to an object \( x \) the temporal logic predicate \( \text{Obj}_{id_x} \text{Op}_{id_y} \) is declared. Notice that the \( \text{Obj}_{id_x} \) prefix is needed in order to distinguish the predicates generated for the same operation but for a different object. Similarly, the predicates for the operation parameters, return values, and attributes are generated. Attributes and return values are translated into TRIO arithmetic variables, which can be integer or real.

The semantics of CDs is fairly limited, because they only declare the shared alphabet between the different diagrams. In particular, only clocks define a behavior that truly belongs to this diagram. The remaining predicates will be constrained according to the rules defined in Section 3.3.4. Given a clock \( c \) with period \( T \), the associated semantics is that its tick event must occur every \( T \) time units. In other words, its clock tick occurs
3.3. Syntax and Semantics

iff it did not tick during the last $T - 1$ time units\(^1\):

$$\text{Lasted}(\neg \text{Clock}_{id_c} \text{Tick}, T - 1) \iff \text{Clock}_{id_c} \text{Tick} \quad (3.1)$$

![Figure 3.4: An example Class Diagram.](image)

Figure 3.4 shows an example CD, where RadarClock ticks every 10 time units.

3.3.2 State Machine

eFT-UML supports State Machines (SMs), each of which is owned by a class, that in turn has a number of instance objects. Each class may have more than one SM. That being the case, the different SMs run in parallel, that is, each object has a current state in each SM and they run independently.

In SM diagrams different states of an object are shown as rectangles with rounded corners, whose labels indicate the name of the states. The initial pseudostate of an SM is shown as a filled circle. Transitions are labeled unidirectional arrows that connect two states, and the current state of an object is traceable following initial pseudostate and event occurrences. Figure 3.5 shows a simple SM diagram of a book object, that has an initial pseudostate, two states, and two transitions that are triggered by operation calls.

![Figure 3.5: An example State Machine Diagram.](image)

The meta-model of transitions in eFT-UML SMs is shown in Figure 3.6. A transition has three optional elements: trigger, guard, and action. Contingent upon true evaluated guard, the trigger forces the transition to take place, whose consequence is the action. A trigger can be either a time constraint or an event, action can be an event or an assignment, and finally the transition can be guarded by a Boolean expression or a time constraint.

\(^1\)TRIO axioms are implicitly asserted for all time instants, hence formula is implicitly interpreted as $\text{Alw(} \text{Lasted}(\neg \text{Clock}_{id_c} \text{Tick}, T - 1) \iff \text{Clock}_{id_c} \text{Tick})$. 

27
Figure 3.6: Meta-model of transitions in eFT-UML State Machines.

![Diagram of transition meta-model]

Figure 3.7: 3.7a: Syntax of transitions in eFT-UML SMs. 3.7b: An example of a transition.

Figure 3.7 shows the syntax of a transition in SMs, together with an example, where the state of corresponding object of the SM changes from Warning to Alert, when the notify operation is invoked and the object has been in the Warning state more than 10 time instants; and the alert operation is invoked afterwards.

To capture all the elements of SMs for an object \( o \) the following predicates according to the SM describing its behavior are generated: For every state \( s \) three predicates \( \text{Obj}_o\text{State}_s\text{Enter} \), \( \text{Obj}_o\text{State}_s\text{Exit} \) and \( \text{Obj}_o\text{State}_s\text{Id} \) are declared, that hold when the object enters into, exits, and stays in the state, respectively. \( \text{Obj}_o\text{Transition}_t\text{Id} \) is declared for every transition \( t \) of an SM, that may have a trigger, a guard, and an action, which are mapped to the following predicates; \( \text{Obj}_o\text{Trigger}_t\text{Id} \), \( \text{Obj}_o\text{Guard}_t\text{Id} \), \( \text{Obj}_o\text{Action}_t\text{Id} \). Finally, \( \text{Obj}_o\text{Invariant}_s\text{Id} \) is declared, if the state has an invariant, which must hold as long as the state is active (\( \text{Obj}_o\text{State}_s\text{Id} \) holds).

The formalization of SMs is fairly intuitive, therefore we will only hint at some of its features. One of the key assumptions in the axiomatization is that all the transitions from a state to another state need one time instant, so multiple simultaneous transitions in SMs is not permitted. Given a state \( s \) owned by an object \( o \), the sets of its incoming and outgoing transitions are defined as \( \text{Incoming}_o \) and \( \text{Outgoing}_o \), respectively. A state
cannot be activated, unless the necessary condition for entering to it is met, which is at least one of incoming transitions took place in the previous time instant:

\[ \text{Obj}_{id}, \text{State}_{id}, \text{Enter} \Rightarrow \]

\[ \text{Past} \left( \bigvee_{t \in \text{Incoming}_{s}} \text{Obj}_{id}, \text{Transition}_{id}, 1 \right) \]  

(3.2)

Having an outgoing transition activated, is the sufficient condition for exiting from a state \( s \) to a destination state \( d \):

\[ \text{Obj}_{id}, \text{Transition}_{id} \Rightarrow \]

\[ \text{Obj}_{id}, \text{State}_{id}, \text{Exit} \land \text{Futr} \left( \text{Obj}_{id}, \text{State}_{id}, \text{Enter}, 1 \right) \]  

(3.3)

This axiom is not generated if \( t \) is a self transition, whose source and destination are same. The object being in the source state and the corresponding guard and trigger evaluated to true are necessary conditions for a transition. An event defines the trigger and the predicate associated to the trigger holds iff the object is in the source state and we have an occurrence of that event. In a similar way, the predicate associated to the guard holds iff its Boolean formula holds. The predicate of action, if there is any, holds at the very same time instant that the transition takes place. Finally, if there are several transitions whose necessary conditions are met, one of them is chosen nondeterministically.

### 3.3.3 Sequence Diagram

A group of objects collaborate in order to fulfill one or more system requirements. These collaborations are described using interaction diagrams, among which the most common and expressive is the Sequence Diagram (SD). Every SD captures a single, possibly complex, scenario where the participant UML elements are two or more objects interacting with one another by passing messages. A lifeline, a vertical dashed line with a label, represents one object participating in the interaction being captured and contains all the relevant events to the corresponding scenario and object. Moreover, events occur according to their vertical geometric position. In other words, chronological order of event occurrence corresponds with their position on the lifeline and an event cannot be preceded by its lower events. In an SD, if object \( \text{obj}_1 \) needs to invoke the operation \( \text{op}_1 \) provided by object \( \text{obj}_2 \), the message is shown as an arrow from \( \text{obj}_1 \) to \( \text{obj}_2 \) labeled by \( \text{op}_1 \). However, \( \text{obj}_1 \) can be equal to \( \text{obj}_2 \), that means the object invokes one of its own operation, that is important to be included to the scenario being captured by the SD. For example, in Figure 3.8, \( \text{radar} \) invokes the operation \( \text{sendSensorDistance} \) that belongs to the \( \text{bus} \) object.

For each SD predicates \( \text{SD}_{id}, \text{Start} \) and \( \text{SD}_{id}, \text{End} \) are generated, that hold at the beginning and the end of the diagram execution, respectively. \( \text{SD}_{id}, \text{Stop} \) predicate is defined and holds when the diagram is interrupted, that is, it has started executing, but it has not reached the end yet. Finally, predicate \( \text{SD}_{id} \) holds as long as the SD is active. \( \text{Msg}_{id}, \text{Start} \) and \( \text{Msg}_{id}, \text{End} \) are declared for every message \( x \), and hold at the time the messages is sent and received, respectively. A message can be sent and received instantly, when these two predicates coincide, or delayed. Additional predicate \( \text{Msg}_{id} \) holds since the message is sent until it is received. \texttt{eFT-UML} allows each SD to define
one parameter whose value is set or chosen nondeterministically at the time the diagram starts and remains constant during the entire execution of the diagram unless there is an assignment meanwhile. With this feature, the user can forward a value between the objects involved in an SD without the need to model complex assignments. For every sequence diagram variable \( y \) found in an SD \( x \), a TRIO arithmetic variable \( SD_{\text{id}} Par_{\text{id}} y \) is declared. A TRIO temporal logic predicate \( Assignment_{\text{id}} \) is declared for every assignment \( x \) in the SD and holds when the assignment takes place.

Time constraints can be included in SDs, that must hold at all time instants the enclosing SD is active. For example, for the time constraint \( x: @SD.\text{End} - @SD.\text{Start} < 21 \), the predicate \( Constraint_{\text{id}} \) is declared that forces the SD to accomplish no longer than 20 time instants. This behavior is asserted by \( SD \Rightarrow Constraint_{\text{id}} \). Figure 3.8 depicts an SD that entails a time constraint that forces the second message to be sent exactly 2 time instants after the first one is sent.

However, passing messages between different objects per se is not enough to capture the behavior of complex systems. Parallel computation, loops, optional, conditional and prohibited behaviors can be captured in UML2 by means of Combined Fragments (CFs). These interactions are core parts of SDs and enabler for designing complex behaviors and scenarios of systems, that has not been covered by \textit{eFT-UML}. In OMG specification, the semantics of CFs are underspecified, therefore they can be interpreted in many ways. The flexible modular formalization of SDs that can also cover CFs is a part of contribution of this thesis that is presented in Section 4.

3.3.4 Shared Events

Different diagrams may share a common set of events that allows different diagrams to communicate with one another. The use of temporal logic to formalize the different diagrams makes this communication straightforward in \textit{eFT-UML}. Shared events include invocation of operations, start, stop, and end of SDs, start and end of IODs, clock ticks, and signals.

For example, we show how SDs and SMs communicate through operation invocations. This communication can be achieved in two complementary ways: SDs and SMs can send operation invocations using messages and actions, respectively. On the other hand, SDs and SMs can listen to operation invocations using found messages and triggers. In

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{sequence_diagram.png}
\caption{An example Sequence Diagram.}
\end{figure}
the model depicted in Figure 3.9, the SD uses a found message to wait for the invocation of operation \( opA \). The operation invocation is sent by the SM while transitioning from State0 to State1, which occurs nondeterministically. The SM then uses a trigger to wait for the invocation of operation \( opB \), which is sent by the message in the SD.

![Diagram](image)

Figure 3.9: The transition from State0 to State1 of Figure 3.9b enables the found message of Figure 3.9a. The message of Figure 3.9a triggers the transition from State1 to State0 of 3.9b.

Diagrams are able to communicate by constraining the combined behavior of the predicates within their formal specifications. Given two diagrams \( d_x, d_y \in D \) that belong to a eFT-UML model, their combined behavior \( \Theta(d_x, d_y) \) is built as follows. Let us define \( \chi : P \times P \rightarrow \{ \text{read, write, nil} \} \) as the function that defines the type of communication between two predicates. \( \chi(p_x, p_y) = \text{read} \) if we have a read relation between \( p_x \) and \( p_y \). \( \chi(p_x, p_y) = \text{write} \) if we have a write relation between \( p_x \) and \( p_y \). Finally, \( \chi(p_x, p_y) = \text{nil} \) if we have no relation between \( p_x \) and \( p_y \). Intuitively if \( \chi(p_x, p_y) = \text{read} \) the diagram that owns \( p_x \) catches an event that occurs in another view, while if \( \chi(p_x, p_y) = \text{write} \) the diagram that owns \( p_x \) forces the occurrence of a certain event in another view. The combined behavior of the different views is built according to the values of function \( \chi \):

- For all \( p_x \in \Gamma(d_x) \) and \( p_y \in \Gamma(d_y) \) such that \( \chi(p_x, p_y) = \text{read} \) we have that \( p_x \Leftrightarrow p_y \) is added to \( \Theta(d_x, d_y) \) and no formulae of the type \( f \Rightarrow p_x \) can be included in \( \Delta(d_x) \).

- For each predicate \( p_y \in \Gamma(d_y) \) that has at least one write relation, let us define \( \text{Reasons}_{p_y} \) as the set of predicates \( p_x \in \Gamma(d_x) \) with \( d_x \neq d_y \) where \( \chi(p_x, p_y) = \text{write} \). Intuitively set \( \text{Reasons}_{p_y} \) contains all the predicates that may cause predicate \( p_y \) to be true. Therefore for each predicate \( p_y \in \Gamma(d_y) \) we generate the following formulae: \( p_y \Leftrightarrow \bigvee_{r \in \text{Reasons}_{p_y}} r \).

As an example let us consider the relations among SMs, SDs and CDs. If \( \text{Obj}_{id_x} \cdot \text{Trigger}_{id_t} \) is the predicate of a trigger waiting for an operation \( x \) in a SM, and \( \text{Obj}_{id_op} \cdot \text{OP}_{id_x} \) is the predicate associated with \( x \) in the corresponding CD, then \( \chi(\text{Obj}_{id_x} \cdot \text{Trigger}_{id_t}, \text{Obj}_{id_op} \cdot \text{OP}_{id_x}) = \text{read} \) since the trigger only holds when the operation holds. By applying the previous definitions, we have that \( \text{Obj}_{id_x} \cdot \text{Trigger}_{id_t} \Leftrightarrow \text{Obj}_{id_op} \cdot \text{OP}_{id_x} \). Notice that
Chapter 3. eFT-UML

the same type of relation holds for found messages in the SDs. In that case, we would have that \( \text{Message}_{id_m} \text{End} \leftrightarrow \text{Obj}_{id_o} \text{OP}_{id_x} \) where \( m \) is a found message in an SD.

If we consider the write relations in which the operation predicate is involved. eFT-UML assumes that an operation holds when one of the messages that instantiate the operation occurs in an SD, or one of the actions that invoke the operation occurs in an SM. In other words, if \( y \) is the name of an operation in a CD, \( \text{Reasons}_y = \text{MessagesOp}_y \cup \text{OperationCallOp}_y \), where \( \text{MessagesOp}_y \) and \( \text{OperationCallOp}_y \) are the set of messages and actions that instantiate \( y \) in the SDs and SMs, respectively. If \( \text{Obj}_{id_o} \text{OP}_{id_y} \) is the predicate associated with \( y \) by applying the previous definitions we have that \( \text{Obj}_{id_o} \text{OP}_{id_y} \Leftrightarrow \bigvee_{r \in \text{Reasons}_y} r \).

3.4 Related Work

Having informal semantics, for the sake of providing leeway for domain-specific refinements, UML has been given many formal semantics during last two decades. Apart from that, there have been also misunderstandings due to the imprecise and inappropriate definition in some parts of UML models [46]. Because of aforementioned reasons, none of given semantics are absolutely correct or incorrect. However, most of the works in literature merely focus on a single diagram type and a concrete semantics to serve its purpose of partially capturing some aspect of a specific domain of interest. A coherent framework that covers a significant number of UML diagrams with flexible formal semantics susceptible of being used in many domains is neglected.

The many works that ascribe semantics to UML can be organized in two groups: those that represent pure theoretical exercises and those whose proposed formalization is executable and susceptible of formal verification. We do not care about the first group, while Motta [63] provides an exhaustive analysis of the members of the second group. This analysis highlights the need for a comprehensive multi-diagram formalization as an enabler for the verification of “significant” UML models.

Diverse attempts to UML formalization can also be categorized from the diagram type point of view, that are the works covering structural diagrams only, behavioral diagrams only, and both types of diagrams. The works on structural diagrams usually focus on checking satisfiability of constraints expressed in OCL (Object Constraint Language). Some well-known examples are the approaches presented in [1, 44, 50]. However, we are interested in the behavioral UML models that are orthogonal to the OCL constraints verification. The focus of the works on behavioral UML models is only on some behavioral diagrams, usually a single diagram type, while neglecting the interdependencies among different diagrams. For example [43, 45] only focus on SMs. Finally, a number of works tried to integrate different UML structural and behavioral views by building a common semantic ground. For instance, in [18] SMs can interact and shape the behavior of the object throughout the system using a ground mathematical semantics. Kuske et al. in [54] show how a central part of UML can be integrated into a single visual semantic model (graphs) to support the visual simulation of integrated specifications. In this category, Foundational UML (fUML) [68] can be mentioned that is strongly influenced by Executable UML [60] and standardized by OMG. fUML provides a foundation for higher-level UML concepts and precise semantics for the parts thereof, by identifying an executable subset of UML2 meta-model. Apart from
3.4. Related Work

the absence of precise mapping between fUML and diagram elements, its semantics cannot be directly executed by any of the available formal verification tools and a further translation step is required.

To compare eFT-UML and the first contribution of this thesis with Kuske [54], Broy [18], and fUML [68], we remark that the focus of our work is not the structure of the UML models (we only consider CDs right now), but rather on their behavior. Our approach has been more practical, since we focus on the creation of a complete verification tool that has been developed incrementally to support a growing number of UML elements, again the underlying metric temporal logic acted as enabler of for this solution.

Micskei and Waeselynck [62] investigated almost all the proposed interpretations of SDs and categorized them based on their semantic choices. Only a few of them have been presented in a way that is rigorous and amenable for formal verification. Fernandes et al. [37] exploit colored Petri nets to explain their semantics, and Knapp and Wuttke [53] propose a translation of SDs into automata. These works only propose single, particular semantics that might yield to inaccurate or even wrong verification results. There must be a formalization based on the domain and enable the designer to the set the interpretations of interest, and carry out the verification accordingly.
Flexible modular formalization of UML models

UML diagrams can be used to model (complex) systems and describe their peculiar aspects. However, their formal verification is not straightforward and rigorous analysis requires that the behavior of the modeling elements be stated unambiguously. To put in other words, the following issues hamper their complete formalization. First, UML is a semi-formal language and its semantics is intentionally underspecified due to providing leeway for domain-specific refinements. Second, in spite of the semantics being underspecified, it is given through textual descriptions and it is often ambiguous.

There have been diverse meanings assigned to UML models. In contrast, the first thread of this thesis does not aim to introduce yet another semantics, but it studies the most significant proposals, organizes them into a single coherent framework, and proposes a solution to interpret Sequence Diagrams (SDs) in a compositional and modular way in order to fulfill OMG’s ambition of keeping UML useful in many domains. Users can decide the interpretations of the key aspects of their interest and the result is a complete and coherent semantics; then, provided some simple constraints are respected to avoid making inconsistent decisions, our framework accommodates all other aspects.

SD is the most expressive diagram type that captures behavioral aspects of the system. Given the importance of SDs and the fact that the elements with many meanings mainly belong to them, this thesis focuses on SDs. However, the same theoretical approach can also be applied to SMs. For this chapter to be self-contained, first we introduce SD with Combined Fragments (CFs) and their informal semantics given by OMG [67]. Then, we present our flexible modular formalization of SDs with respect to the most important semantic variation points.

The chapter is structured as follows. Section 4.1 presents the meta-model and structure of an SD and also describes the informal semantics of different types of CFs.
Chapter 4. Flexible modular formalization of UML models

Based on OMG specification. Section 4.2 explores ambiguities in SD semantics, extracts variation points, and proposes legitimate semantic variants. The basic modular semantics are presented in Section 4.3, that are required for the semantics of CFs and SDs. Finally, Section 4.4 presents the main parts of the flexible modular semantics of diverse CFs. The complete formalization is presented in detail in Appendix A.

4.1 SD and Combined Fragments

SDs capture the control flow of systems, interactions amongst different objects engaged in a specific scenario, and what happens to the involved objects since their creation (or the start of their interaction) until their destruction (or the end of their contribution in the scenario). Lifelines and messages are the main elements of SDs in UML1. For each object participating in a scenario, a lifeline stands for the event occurrences related to the object. Sending and receiving messages, instantiation/creation and destruction of the corresponding object are examples of these events. However, the types and complexity of scenarios being captured by SDs in UML1 are very limited due to the lack of expressive elements. More precisely, the scenarios could be essentially a sequence of simple event occurrences.

There was a great enhancement from UML1 to UML2 for SDs. Combined Fragments (CFs) were added to SD, and their semantics and meta-model were rewritten accordingly, that increased the expressiveness of this type of diagram. Currently, SDs are similar to programming languages and in some cases even more expressive. This is owed to the ability of CFs in capturing complex behaviors, like optional and alternative interactions, parallel execution, and iterations.

CFs are elements that can capture a specific type of behavior. They allow the user to
model complex interactions among objects in a succinct and organized manner. Each CF depends on an interaction operator: `alt`, `opt`, `par`, `loop`, `break`, `seq`, `neg`, and others [67]. Every CF comprises one or more operands (OPs) and associated Interaction Constraints/guards (nothing means true). Each operand, in turn, contains messages—and further CFs—that are processed according to the interaction operator associated with the enclosing CF. Figure 4.1 depicts the important parts of CF meta-model, defined in OMG specification [67]. Figure 4.2 depicts an example SD namely `SD1` that captures the scenario employing an alternative (`alt`) CF containing two OPs. There are two objects engaged in this interaction for which there exist two lifelines, and every event related to an object must be initiated or received by the corresponding lifeline.

In the following, we briefly introduce the most significant CFs and their semantics based on OMG [67].

- **Alternative CF – `alt`**
  The alternative CF chooses an interaction to pursue, among several interactions, regarding the current situation. It has one or more OPs, each of which may have a guard. At most one of the OPs is chosen that has no guard (that means an implicitly true) or its guard is evaluated to true. If none of the OPs are allowed to be executed, the whole alternative CF is ignored and the SD moves to the elements after the CF. However, an alternative CF may have an additional OP, that is `else` OP. It gets activated, iff all the other OPs do not have a chance of getting activated due to their guards being evaluated to false. For example, in the alternative CF shown in Figure 4.3, since both OPs have implicit true guard, either of them can be executed, but not both. In other words, `{⟨!?m1⟩}` and `{⟨!?m2⟩}` are valid, but not `{⟨!?m1⟩, ⟨!?m2⟩}`. The notation we use decomposes a message in the two events that correspond to sending (!) and receiving it (?). The shortcut `!?m` is equivalent to `!m` and `?m` at the same time instant, and a pair of ⟨⟩ enclose the events that happen at the same time.
Chapter 4. Flexible modular formalization of UML models

Figure 4.3: An example Alternative Combined Fragment.

• **Optional CF** – $\text{opt}$
  The optional CF has only one OP. The guard of its sole OP is the necessary condition for its activation. In other words, if the OP is activated, it means that its guard is evaluated to true, nevertheless, having a true evaluated guard does not mean that the OP for sure gets activated (true evaluated guard is not a sufficient condition) and it may get ignored. It might be counter-intuitive, but this CF demonstrates a nondeterministic behavior over whether or not a true guarded interaction is activated. Optional CF is a special case of alternative CF. It can be built using an alternative CF that has two OPs: one is equal to the optional OP and the other one is empty. For example, in the optional CF shown in Figure 4.4, since the OP has implicit true guard, it may get activated. In other words, both $\{\langle \mathbf{!m1} \rangle\}$ and $\{\langle \mathbf{!m1} \rangle, \langle \mathbf{!m2} \rangle\}$ are valid.

Figure 4.4: An example Optional Combined Fragment.

• **Parallel CF** – $\text{par}$
  Parallel CF demonstrates a behavior of parallel execution of several OPs. This CF has two or more (possibly) guarded OPs, each of whose sufficient condition for activation is the corresponding guard being evaluated to true at the time of CF activation. For example, in the parallel CF shown in Figure 4.5, since the both OPs have implicit true guard, they get activated, and the events in different OPs can interleave. In other words, $\{\langle \mathbf{!m1} \rangle, \langle \mathbf{!m2} \rangle\}$, $\{\langle \mathbf{!m2} \rangle, \langle \mathbf{!m1} \rangle\}$,
4.1. SD and Combined Fragments

\{⟨!m1⟩, ⟨!m2⟩, ⟨?m2⟩ ⟨?m1⟩\}, and \{⟨!?m2, !m1⟩, ⟨?m1⟩\} are valid.

• Loop CF – loop
This CF captures the behavior of iteration with nondeterminism in the number of iterations. It has one and only one OP, whose interaction constraints entails an optional guard, minimum and maximum number of iterations, in the format of guard[min, max]. Loop CF iterates at least min times, even if the guard is evaluated to false, and at most max times. The true evaluated guard is the necessary (not sufficient) condition of iterating more than min times. In other words, if the guard is evaluated to true, the loop CF may or may not enter the iteration number min+1 and the next ones. For example, in the loop CF shown in Figure 4.6, since the OP has implicit true guard, 2 as min, and 3 as max, the CF can iterate either 2 or 3 times, so \{⟨!?m1⟩, ⟨!?m2⟩\}, and \{⟨!?m1⟩, ⟨!?m2⟩ ⟨!?m1⟩, ⟨!?m2⟩\} are valid.

• Break CF – break
This CF represents a breaking scenario, such that its content gets executed instead of the rest of enclosing interaction fragment, contingent on its guard being evaluated to true. It has only one OP, possibly with a guard. The absence of guard means implicit true, but in this CF, an implicit true guard does not necessary guarantee activation of the CF. In other words, if there is no guard, the choice between entering
break and pursuing the rest of enclosing interaction is made nondeterministically. For example, in the SD shown in Figure 4.7, since the break CF has implicit true guard, \{\langle !?m1 \rangle \} and \{\langle !?m2 \rangle \} are valid, but not \{\langle !?m1 \rangle, \langle !?m2 \rangle \}.

![Figure 4.7: An example Break Combined Fragment.](image)

### 4.2 Semantic Variation Points and Variants

The OMG specification [67] does not define UML SDs semantics in detail to foster their adoption in different domains, but this allowed for diverse interpretations: for example, Micskei and Waeselynck [62] survey thirteen (sometimes similar) different alternatives.

We use the diagram of Figure 4.8 (SDSearch) to illustrate interpretations, explore ambiguities, and identify possible solutions. Informally, app is given a list of keywords, it pings server1 and server2 in parallel and passes the keywords to the first server that replies. app also periodically updates the screen with the results obtained from the server. Most of the problems come with the interpretation of CFs.

The first problem we address concerns the role played by the borders of a CF. They can act as mere (graphical) containers of operands, whose execution order can then cross the borders, or be used to constrain the execution sequence of the messages of the different operands based on their position in the diagram. For example, in this latter case, it would not be possible for replyS1 to be sent before pingS2 is received. More rigorously\(^1\), \(T_1 = \{\langle !\text{pingS1} \rangle, \langle !\text{pingS2} \rangle, \langle !\text{replyS1} \rangle, \langle ?\text{pingS2} \rangle \} \) would be a trace fragment that satisfies the semantics of the diagram according to the former interpretation, but it would violate the latter since \(!\text{replyS1} \) cannot happen before \(?\text{pingS2} \).

The more general question is thus how the executions of the different CFs are blended ([Combine] in Figure 4.9). The OMG states that: “The semantics of an interaction operand is given by its constituent interaction fragments combined by the implicit seq operation”. This means that the combined semantics is defined through the semantics of seq, which obliges fragments to combine operands through weak sequencing (WS in Figure 4.9) that is defined as the following:

- If two events refer to the same lifeline, they follow the top-down order imposed by the lifeline.

---

\(^1\)The notation we use decomposes a message in the two events that correspond to sending (!) and receiving it (?). The shortcut !?m is equivalent to !m and ?m at the same time instant, and a pair of \(\langle \rangle \) enclose the events that happen at the same time.
4.2. Semantic Variation Points and Variants

- If they refer to different lifelines whose corresponding objects exchange messages (for example, pingS1 and replyS1 between app and server1), then the send event on the first lifeline must also come before the receive event on the second lifeline. Hence, the order on a lifeline partially depends on the order of exchanged messages.

- Finally, if two events refer to independent lifelines, that do not exchange messages, they can appear in any order (for example, replyS1 and replyS2 can happen in any order).

The OMG also states that borders do not impose any ordering constraint and that the events before a CF may happen after it and the events after the CF may happen before it. [62] highlights that several proposed semantics, for example, the work by Fernandes et al. [37], do not comply with what the OMG states. They impose an additional constraint on the weak sequencing that forces a synchronous composition (SYNC) among the different elements: the sequence of event occurrences respects the borders of CFs; that means, events before a CF happen before it and events after the CF happen after its conclusion. This means that OMG would allow $T_1$ to be a fragment of a correct execution trace, while this further constraint would forbid it.

CFs can also embed loops, but how the events in the different iterations are combined ([Loop]) is questionable. The occurrences in one iteration can: (a) be strictly separated from or (b) be interleaved with those of the others. Thus, the
Chapter 4. Flexible modular formalization of UML models

The correctness of $T_2 = \{\langle \text{searchNextS}_1^{i1} \rangle, \langle \text{resultS}_1^{i1} \rangle, \langle \text{updateResultS}_1^{i1} \rangle, \langle \text{searchNextS}_1^{i2} \rangle, \langle \text{resultS}_1^{i2} \rangle, \langle \text{updateResultS}_1^{i1} \rangle, \langle \text{updateResultS}_1^{i2} \rangle, \langle \text{updateResultS}_1^{i2} \rangle \}$, where $iX$ means the $x$th iteration, is questionable.

The OMG states that the semantics of loop is equivalent to the recursive application of the semantics of seq (WS). These loops then become more permissive than those in programming languages — since their iterations cannot be intertwined. If one needed to mimic them, this would not be possible, in line with the OMG specification, even by using general ordering$^2$ in loop. Knapp and Wuttke [53] proposed to have weak sequencing for all fragments but loops. If iterations are to be combined synchronously (SYNC), $T_2$ would not be a correct execution trace while it would be acceptable according to the pure OMG specification.

Even the evaluation of guards comes with some problems ([When]). One may say that (a) all lifelines evaluate guards at a single global time (SGT), or (b) each lifeline does it at particular independent local times (ILT). For example, $T_1$ corresponds to the case in which app and server1 interact through the first operand of the alt fragment, and server1 replies before server2 has received the ping request. The correctness of this sequence depends on how the execution of composed fragments is interpreted: synchronous composition imposes that all guards be evaluated by each lifeline when the execution of the CF starts. Weak sequencing would allow the first lifeline entering the CF to evaluate guards and decide which operand to execute if needed. All other lifelines must then obey the decision taken by the first lifeline, even if meanwhile guards have changed.

Besides the evaluation time, more than one guard may be true in an alt fragment ([Choice]). If this is the case, one could: (a) pick one nondeterministically (ND), or (b) select the first true guarded operand from the top (FFT). The OMG only states that “At most one of the operands will be chosen”, but there is no advice on the selection in the case of alternatives. Most of the known interpretations simply pick one true-guarded operand nondeterministically. Others simply select the first one from the top. The latter choice is the way to go if one adopted the implicit assumption that operands are ordered top-down, according to their priority.

All the aforementioned problems and choices must be properly addressed, and one single solution be identified. There might be different ways of presenting a scenario by means of SDs, but there must always be one and only one interpretation of the diagram.

To this end, the decision tree of Figure 4.9 summarizes our analysis and identifies six possible coherent and complete interpretations given the different choices and their mutual constraints. Gray nodes identify inconsistent interpretations, and therefore, the derivation is aborted. For example, a coherent interpretation cannot use SGT and WS: a single global time cannot be selected when fragments are composed according to weak sequencing. Similarly, WS for loops is not compatible with SYNC, as composing fragments through synchronous composition and iterations in loops through weak sequencing would be incoherent.

Of the six resulting configurations, Config5 is the only one that fully complies with OMG’s specification. Config3 and Config4 are special cases: they use a single global time to evaluate the guards associated with loops, and independent local times on each lifeline for the other fragments.

---

$^2$A general ordering is a binary relation between events that describes that an event must occur before another in a trace.
4.3 Basic Modules

In this section, the most important and frequently used modules in our formalization that capture cross-cutting aspects of the semantics of SDs are presented. It is worthy to mention that, these modules are not meant to be used only for SDs; rather, they can also be used in other parts of eFT-UML or even in the formalization of other timed models.

4.3.1 Order

The formalization of the ordering between two events is the simplest part of the semantics, and it is the building block for formalizing partial and general ordering in SDs. However, based on the exact description the formal semantics can be different. Even the simple notion of “\( ev_1 \) is followed by \( ev_2 \)” within the description raises some questions about the validity of traces. Is occurrence of \( ev_1 \) a necessary condition for occurring \( ev_2 \) (Is \( ev_1 \) alone valid?), or sufficient condition? Is \( ev_1 \) the only event that can trigger \( ev_2 \), in other word, would it be a valid trace if we had \( ev_2 \) without a previous \( ev_1 \)? Can they occur at the same time? Several interpretations are possible. In the following, three types of the order together with their formal semantics are presented that enables us to precisely capture the order of event occurrences.

1. Mono-Directional Order:

   Event \( ev_2 \) may be triggered by the occurrence of event \( ev_1 \) under certain circumstances. In this case, we formalize that, after \( ev_1 \), \( ev_2 \) occurs, provided that a suitable guard holds at the moment \( ev_1 \) occurs. However, if the ordering is only mono-directional, an occurrence of \( ev_2 \) does not imply a previous \( ev_1 \). Formula
(4.1) captures this notion of ordering (where \textit{guard} and \textit{exception} are placeholders for subformulae that are introduced below).

\[
ev_1 \land \text{guard} \Rightarrow \left( \text{Until}_{ei}((\neg ev_1 \land \neg ev_2), \text{exception}) \lor \text{Until}_{ei}((\neg ev_1 \land \neg \text{exception}), ev_2) \right)
\] (4.1)

The next formula, instead, defines that event \( ev_2 \) cannot occur at the same time as its trigger \((ev_1)\):

\[
ev_1 \land \text{guard} \Rightarrow \neg ev_2.
\] (4.2)

Following [6], we write the modular semantics of SDs to fit it into a larger UML semantics, which also includes other diagrams such as Interaction Overview Diagrams. Hence, we allow an SD to be stopped at any time during its execution, provided a suitable event (an \textit{exception}, for example, a timeout) occurs. This is captured by Formula (4.1), which states that \( ev_2 \) occurs after \( ev_1 \land \text{guard} \) holds, unless an exception intervenes. Finally, we introduce the following abbreviation:

\[
\begin{align*}
\text{OrderMonoD}(ev_1, ev_2, \text{guard}, \text{exception}, \text{isConcurrent}) & \overset{\text{def}}{=} \\
& \begin{cases} 
(4.1) & \text{if isConcurrent = true} \\
(4.1) \land (4.2) & \text{otherwise}
\end{cases}
\end{align*}
\]

2. Mono-Directional Reverse Order:

When event \( ev_2 \) is necessarily triggered by event \( ev_1 \), but not all occurrences of \( ev_1 \) produce \( ev_2 \), then we formalize that the former has to be preceded by the latter, but not the converse. This is captured by the following formula:

\[
ev_2 \Rightarrow \text{Since}_{ei}((\neg ev_2 \land \neg \text{exception}), (ev_1 \land \text{guard})).
\] (4.3)

Abbreviation \text{OrderMonoDRev} is similar to \text{OrderMonoD} and is omitted for brevity.

3. Bidirectional Order: This is the conjunction of the two previous cases: \( ev_1 \) occurs if, and only if, \( ev_2 \) also occurs later (unless an exception occurs before):

\[
\begin{align*}
\text{Order}(ev_1, ev_2, \text{guard}, \text{exception}, \text{isConcurrent}) & \overset{\text{def}}{=} \\
& \text{OrderMonoD}(ev_1, ev_2, \text{guard}, \text{exception}, \text{isConcurrent}) \land \\
& \text{OrderMonoDRev}(ev_1, ev_2, \text{guard}, \text{exception}, \text{isConcurrent})
\end{align*}
\]

4.3.2 Borders

Each SD, CF, lifeline, operand of CF and message has a beginning, an end, and a duration throughout which the element is “active” (for example, a message that was sent and has yet to be received, or an operand that was entered and not exited); hence, we treat them as “modules”, each with its own semantics. For each module \( M \) we introduce three predicates \( M, M_{\text{Start}} \) and \( M_{\text{End}} \) capturing, respectively, the module being “active”, its start, and its end. Formulae (4.4)-(4.5) formalize their behavior.

\[
M \leftrightarrow \text{Since}_{ei}(\neg M_{\text{End}} \land \neg \text{exception}, M_{\text{Start}})
\] (4.4)

\[
M_{\text{Start}} \leftrightarrow \text{Since}_{ei}(\neg M \land \text{exception}, M) 
\] (4.5)
4.4 Combined Fragments

\[ M_{\text{Start}} \Rightarrow \text{Until}_i(\neg M_{\text{Start}}, M_{\text{End}} \lor \text{exception}) \]  \hspace{1cm} (4.5)

Formula (4.4) defines that a module is active from its start until its end (which could occur with an exception in the enclosing module), and Formula (4.5) states that after a module starts, it must end, possibly because of an exception, and there is no further start until then. Abbreviation Borders captures the semantics of the start and end of a module:

\[ \text{Borders}(\text{Module}, \text{exception}) \overset{\text{def}}{=} (4.4) \land (4.5). \]

4.3.3 Auxiliary Operators

It is useful to introduce, as abbreviations, variations of the “eventually” TRIO operators (SomF and its past counterpart SomF) that have the meaning of “eventually during the current execution of the enclosing fragment” (which could be a CF or possibly the whole SD). We call these abbreviations SomFin and SomPin, where the subscript \( i \) means that the current instant is included. They are defined as follows:

\[ \text{SomFin}(ev_1, \text{enclosingModule}) \overset{\text{def}}{=} \neg \text{Until}_{ii}(\neg ev_1, \text{enclosingModule}_{\text{End}}). \]
\[ \text{SomPin}(ev_1, \text{enclosingModule}) \overset{\text{def}}{=} \neg \text{Since}_{ii}(\neg ev_1, \text{enclosingModule}_{\text{Start}}). \]

4.4 Combined Fragments

As Figure 4.9 shows, to define the semantics of SDs there are four choices to make. In the rest of this section we describe how these choices impact different pieces of the formal semantics, and we show how they are combined to form a complete semantics of diagrams. We mostly focus on the alt combined fragment, and provide highlights of par and loop. For break CF, we both formalize standard semantics and improve its expressiveness in comparison with the one in the UML. The complete formalization of alt, opt, par, loop, and break are presented in Appendix A.

Alternative

Let us consider an alt fragment Alt that is part of a SD \( S \). We first consider the semantics when the choices for [Combine] and [Choice] are, respectively, WS and ND. Then, we show how the semantics changes for different choices. The following formula —introduced in Section 4.3.2 —defines the basic behavior of the start and end propositions of Alt, and it identifies \( S_{\text{Stop}} \) as the event that can interrupt the execution of the combined fragment.

\[ \text{Borders}(\text{Alt}, S_{\text{Stop}}) \]

Formula (4.7) defines that fragment Alt is activated as soon as one of the \( n \) lifelines it spans (which are indicated in the formula by \( \text{Alt}_L^i \), with \( 1 \leq i \leq n \)) enters the fragment (which is represented by proposition \( \text{Alt}_L^{i}_{\text{Start}} \)).

\[ \left( \bigvee_{i=1}^{n} \text{Alt}_L^{i}_{\text{Start}} \Rightarrow \text{Alt} \right) \land \left( \text{Alt}_{\text{Start}} \Rightarrow \bigvee_{i=1}^{n} \text{Alt}_L^{i}_{\text{Start}} \right) \]  \hspace{1cm} (4.7)
To capture the different semantics, we introduce propositions to represent when a lifeline $L_j$ enters the combined fragment $\text{Alt} (Alt_{L_j}^{\text{Start}})$ and when it enters one of its $m$ operands $\text{OP}_i (Alt_{\text{OP}_i}^{\text{L}_j} L_j^{\text{Start}})$. The two events are of course related, as entering operand $\text{OP}_i$ occurs if, and only if, the combined fragment was previously entered. This is captured by Formula (4.8), in which the guard $Alt_{\text{OP}_i}$ implies that the lifeline can enter only the active operand; that is, lifelines that enter the combined fragment at different instants must enter the same operand.

$$\bigwedge_{i=1}^{m} \bigwedge_{j=1}^{n} \text{Order}(Alt_{L_j}^{\text{Start}}, Alt_{\text{OP}_i}^{\text{L}_j} L_j^{\text{Start}}, Alt_{\text{OP}_i}, S_{\text{Stop}}, \text{true})$$

Formula (4.9), instead, defines that the exit operand $\text{OP}_i$ of a lifeline $L_j$ of fragment $\text{Alt}$ is followed by $L_j$ exiting $\text{Alt}$ itself.

$$\bigwedge_{i=1}^{m} \bigwedge_{j=1}^{n} \text{OrderMonoD}(Alt_{\text{OP}_i}^{\text{L}_j} L_j^{\text{End}}, Alt_{L_j}^{\text{End}}, \text{true}, S_{\text{Stop}}, \text{true})$$

Formula (4.10) captures the semantics of the $\text{else}$ operand of $\text{Alt}$, which is entered iff the guards of all other operands are false when the combined fragment is entered.

$$Alt_{\text{OP}_i}^{\text{Else}} \text{Start} \iff (\lnot(\bigvee_{i=1}^{m} Alt_{\text{Guard}_i}) \land Alt_{\text{Start}})$$

Formula (4.11) defines that when combined fragment $\text{Alt}$ starts being executed, exactly one of its operands (possibly the $\text{else}$ operand) starts its execution.

$$Alt_{\text{Start}} \Rightarrow \bigvee_{i \in [1,m] \cup \text{Else}} (Alt_{\text{OP}_i}^{\text{Start}} \land \bigwedge_{j \in [1,m] \cup \text{Else}} \lnot Alt_{\text{OP}_j}^{\text{Start}})$$

Formula (4.12) formalizes that the combined fragment ends when one of its operands (which, by the constraints above, must be the one that was chosen for execution) terminates.

$$\bigvee_{i \in [1,m] \cup \text{Else}} Alt_{\text{OP}_i}^{\text{End}} \iff Alt_{\text{End}}$$

Finally, Formula (4.13) states that an operand is chosen only if its guard is true when $\text{Alt}$ starts its execution.

$$\bigwedge_{i=1}^{m} (Alt_{\text{OP}_i}^{\text{Start}} \Rightarrow Alt_{\text{Start}} \land Alt_{\text{Guard}_i})$$

The semantics of the $\text{Alt}$ combined fragment is completed by adding the formulae defining the behavior of each operand, but we skip this for brevity.

Finally, we define the following abbreviation (where $F_i$ is the $i$th formula):

$$\text{AltCF}(\text{Alt}, WS, ND) \overset{\text{def}}{=} \bigwedge_{i \in [4.6.4.13]} \text{Alw}(F_i).$$
4.4. Combined Fragments

If the user selects \textbf{FFT} as semantic choice instead of \textbf{ND}, Formula (4.11) is replaced by the following one, which selects an operand \textit{OP}_i only if its guard holds, and \textit{Guard}_j is false for all \( j < i \):

\[
\bigwedge_{i=1}^{m} \text{Alt}_\text{OP}_\text{Start} \iff \text{Alt}_\text{Start} \land \text{Alt}_\text{Guard}_i \land \neg \bigvee_{j=1}^{i-1} \text{Alt}_\text{Guard}_j \tag{4.14}
\]

In this case, we have the following abbreviation:

\[
\text{AltCF}(\text{Alt}, \text{WS}, \text{FFT}) \overset{\text{def}}{=} \bigwedge_{i \in [4,6,4,10] \cup [4,12,4,14]} \text{Alw}(F_i).
\]

If the choice of \textbf{[Combine]} is \textbf{SYNC} instead of \textbf{WS}, the admissible traces are a subset of those allowed by \textbf{WS}, since there are additional constraints on the start and end of the combined fragment. More precisely, in this case, the following formula is added:

\[
(\text{Alt}_\text{Start} \iff \bigwedge_{i=1}^{n} \text{Alt}_\text{L}_\text{Start}^i) \land (\text{Alt}_\text{End} \iff \bigwedge_{i=1}^{n} \text{Alt}_\text{L}_\text{End}^i) \tag{4.15}
\]

and the corresponding abbreviation becomes the following:

\[
\text{AltCF}(\text{Alt}, \text{SYNC}, \text{ND}) \overset{\text{def}}{=} \bigwedge_{i \in [4,6,4,13] \cup \{4.15\}} \text{Alw}(F_i).
\]

Another possibility to define \text{AltCF}(\text{Alt}, \text{SYNC}, \text{ND}) is to replace each proposition \textit{Alt}_\text{L}_\text{Start}^i (resp. \textit{Alt}_\text{L}_\text{End}^i) with \textit{Alt}_\text{Start} (resp. \textit{Alt}_\text{End}), and to eliminate Formula (4.7), which is in this case subsumed by the others.

**Parallel**

For a \textit{par} combined fragment the only meaningful semantic choice is between \textbf{WS} and \textbf{SYNC}. As mentioned above, the semantics of \textbf{SYNC} can be obtained by adding constraints to the \textbf{WS} semantics, so we focus on the latter.

When a \textit{par} combined fragment \( P \) is activated (i.e., when one of the lifelines enters the fragment first) the guards of its operands are evaluated and for those that are true the corresponding operands are activated. When another lifeline enters the fragment, it is allowed to start executing its events in the activated operands in parallel. If all guards evaluate to false, the \textit{par} fragment collapses over all the lifelines, that is, its start coincides with its end, as defined by Formula (4.16).

\[
P_{\text{Start}} \land \neg \bigvee_{i=1}^{m} \text{P}_\text{Guard}_i \Rightarrow \bigwedge_{i=1}^{n} \text{SomFIn}_i \left( P_{\text{L}_\text{Start}^i \land P_{\text{L}_\text{End}^i}, P} \right). \tag{4.16}
\]

Conversely, Formula (4.17) states that the end point of a lifeline in a \textit{par} fragment should occur either at the same time of its start point (in case all guards are false), or when it leaves the last operand of the fragment.

47
Chapter 4. Flexible modular formalization of UML models

\[ \bigwedge_{i=1}^{n} (P_{L_{\text{End}}} \Rightarrow (P_{L_{\text{Start}}} \land \text{SomPIn}_{i}(P_{\text{Start}} \land \neg \bigvee_{j \in [1,m]} P_{\text{Guard}}_{j}, P)) \lor \bigvee_{j \in [1,m]} P_{\text{OP}}_{j} \land \bigwedge_{j \in [1,m]} (P_{\text{OP}}_{j} \Rightarrow P_{\text{OP}}_{j} \land \text{End})) \] (4.17)

The full semantics of fragment \( P \) is then obtained by the conjunction of Formulae (4.16) and (4.17), plus others similar to those presented in Section 4.4.

Loop

A loop fragment with a min and a max number of iterations has the same behavior as a min number of seq fragments (i.e., it would behave according to the sequential semantics of SDs), followed by a max – min number of opt fragments (which are a special case of the alt fragment). Each of the above fragments contains the same operand, indicated in the following as \( \text{Loop\_OP} \), which is repeatedly activated during the loop. The options for the semantics of loops (see Figure 4.9) are WS or SYNC.

We build the WS semantics by reducing a loop to a sequence of seq and opt fragments, as explained above. Since in a WS semantics different lifelines can be in different iterations of the loop at the same time (for example, lifeline \( L_{1} \) is in iteration 1, but \( L_{3} \) is in iteration 2), we introduce different sets of predicates for each iteration. Then, the semantics of each iteration is produced similarly to the case presented in Section 4.4, using its specific propositions. In addition, the end events of each iteration are linked to the start events of the next one by the ordering mechanisms of Section 4.3.1.

When the chosen semantics for loops is SYNC, the formalization can be simplified. Since all lifelines execute the same iteration at the same time, we introduce only one set of propositions for operand \( \text{Loop\_OP} \). We keep track of the current iteration through propositions that describe a finite counter \( C \), whose values range from 0 to max.

For example Formula (4.18) defines that if, at the end of an iteration, the guard of the loop is false and the number of iterations is over or at the minimum, then the execution of the loop ends.

\[ \text{Loop\_OP}_{\text{End}} \land C \geq \text{min} \land \neg \text{Loop\_Guard} \Rightarrow \text{Loop}_{\text{End}} \] (4.18)

Formula (4.19) states that, if the guard holds at the end of the iteration, we have a nondeterministic choice of continuing with the loop or ending it (\( \oplus \) is the exclusive or).

\[ \text{Loop\_OP}_{\text{End}} \land C \geq \text{min} \land C < \text{max} \land \text{Loop\_Guard} \Rightarrow \text{Futr}(\text{Loop\_OP}_{\text{Start}}, 1) \oplus \text{Loop}_{\text{End}} \] (4.19)

Finally, Formula (4.20) states that when the maximum number of iterations is reached, the execution ends.

\[ \text{Loop\_OP}_{\text{End}} \land C = \text{max} \Rightarrow \text{Loop}_{\text{End}} \] (4.20)

The complete semantics of the loop fragment is produced from the basic building blocks in a similar way as before.
4.5. Integrated Modular Semantics

Break

This CF has only one OP and if its guard is evaluated to true, after processing all the events inside its sole OP, the rest of enclosing fragment will be ignored. For break, we propose a new semantics that allows this CF to break more than one border. Currently, in UML it can break one border, that means it can ignore the rest of only its enclosing fragment. For example, if it is in a loop and gets activated, the rest of current iteration will be ignored, not the whole loop CF. We found it necessary to improve its expressiveness proposing an additional parameter namely JumpLength, that determines how many borders break must break.

Suppose that there is a break inside a loop. When JumpLength is one, rest of iteration gets ignored and subsequently next iteration starts (similar to continue in programming languages). When it is two, the rest of iteration, and the whole loop CF is ignored (similar to break in programming languages). When it is three, the rest of iteration, the whole loop, and also its enclosing fragment are ignored, and so on for the higher values. However, our semantics provide an additional feature that can be ignored by assigning one to JumpLength, that makes our break compliant with OMG specification.

The main idea of formal semantics capturing the behavior of this CF is to introduce additional predicate for each lifeline in each fragment, namely skip. If the predicate \( F_x L_1 \_Skip \) holds, the rest of events on the lifeline \( L_1 \) within the borders of \( F_x \) are ignored, and consequently \( F_x L_1 \_End \) must hold. To determine how many of these skip predicates must be activated for upper levels, we have a list of enclosing fragments for each break. Considering the previous example, this list will be \{SD_1 \_Loop_1 \_OP, SD_1 \_Loop_1, SD_1\}. If JumpLength is one, only the skip predicates of lifelines in the first element of the list will be activated, if two those for the first two elements, and if three the skip predicates of all lifelines of three elements will be activated. In the latter case, the rest of whole SD_1 will be ignored after leaving the break CF.

4.5 Integrated Modular Semantics

At the basis, given an SD \( S \), we introduce a set of logic propositions that capture the behavior of the elements of \( S \). For example, proposition \( S_{Start} \) describes the start of the execution of diagram \( S \). If \( S \) includes a message \( m \) that is exchanged between two objects, propositions \( m_{Start} \) and \( m_{End} \) correspond to the message being respectively sent and received, and so on. Every element of \( S \) that is relevant for its semantics has a corresponding proposition, including the borders of combined fragments (to represent when they are crossed during the execution of \( S \)), their guards, etc. Then, for each aspect of the semantics of an SD \( S \), we introduce logic formulae that capture it. For example, given a lifeline \( L \) of \( S \), we introduce a set of formulae, say \( Ord_L \), that capture the ordering of events along the lifeline. If \( L_1 \ldots L_n \) are all the lifelines of \( S \), the complete formalization of the ordering of events through \( S \) is given by \( Ord_S = \bigwedge_{i=1}^n Ord_{L_i} \).

Now, consider a combined fragment \( C \) that spans different lifelines. A set of formulae, say \( Sem_C \), captures the semantics of \( C \), including the conditions under which \( C \) is entered (for example, whether all lifelines enter the fragment at the same time or not) or exited. Formula \( Ord_S \land Sem_C \) constrains the ordering of events to obey the semantics of \( C \).
Chapter 4. Flexible modular formalization of UML models

Note that formulae $Ord_S$ and $Sem_C$ will include common propositions such as, for example, the proposition corresponding to $C$ being entered, since it is also an event that occurs along the lifelines. These shared propositions provide a form of “synchronization” among the various parts of the formalization, as their behavior is constrained by several concurring aspects.

The different semantic variants presented in Section 4.2 give rise to different ways of formalizing the various aspects of an SD $S$. For example, the semantics of combined fragment $C$ is different depending on whether a $SYNC$ or a $WS$ semantics is used. If we indicate by $Sem_C^{SYNC}$ the semantics of $C$ in the former case and by $Sem_C^{WS}$ the one in the latter, then changing from one to the other is as simple as substituting $Sem_C^{WS}$ for $Sem_C^{SYNC}$ in $Ord_S \land Sem_C^{SYNC}$. 
Bit-vector-based Encoding for LTL and CLTL

The goal of this thesis is to efficiently verify UML/eFT-UML models. So far, we introduced a coherent framework to formalize UML models with many meanings. Up to this point, we have a huge TRIO reducible to LTL/CLTL formula whose satisfiability must be checked employing a verification technique. The importance of verification techniques to be efficient rises from the fact that almost all the verification duration is spent for the satisfiability checking, while the duration of translating UML models is almost negligible.

The state explosion has always been a crucial problem in satisfiability checking of LTL formula, rather generally, formal verification. With respect to our problem, where the formal verification of UML models is reduced to satisfiability checking of LTL formulae, as the UML model becomes more complex, the resulting LTL formula grows in size accordingly and makes the whole verification process slow, and at a given point impossible. The second contribution of this thesis is bit-vector-based encodings for these LTL and CLTL formulae, which have allowed us to move a significant step forward in tackling the scalability of their formal verification.

In this chapter, first, Quantifier-Free fixed-size Bit-vector Logic (QF_BV) is briefly introduced in Section 5.1. Then, our novel bit-vector-based encoding of LTL formulae is presented together with its correctness proof in Sections 5.2-5.4. Finally, without going into much formal details, we present our CLTL encoding whose corner stone is our bit-vector-based encoding for plain LTL formulae in Section 5.5.

5.1 Quantifier-Free fixed-size Bit-vector Logic

In this section, we briefly present the operations on bit-vectors that are defined in QF_BV and that we use in our encoding.
Chapter 5. Bit-vector-based Encoding for LTL and CLTL

A bit-vector is an array whose elements are bits (Booleans). In Bit-vector Logic, the size of a bit-vector (number of bits) is finite, and can be any nonzero number in \( \mathbb{N} \). For the bit-vector \( \bar{x} \) with size \( n \), we use the notation \( \bar{x}[n] \), or simply \( \bar{x} \) when the size is not important or can be inferred from the context. Furthermore, \( \bar{x}[i] \) stands for the \( i^{th} \) bit in the bit-vector \( \bar{x} \), where bits are indexed from right to left. Accordingly, \( \bar{x}[n-1] \) is the leftmost and most significant bit, and \( \bar{x}[0] \) is the rightmost and least significant bit. For constants we use the notation \( \bar{c}[n] \), which is the two’s complement representation of integer \( c \) over \( n \) bits, for example, \( \bar{2}[4] \) is 1110.

QF_BV offers a wide range of operators, Table 5.1 introduces some of the operators that we use in our encoding.

**Table 5.1: Signature of used QF_BV operators.**

<table>
<thead>
<tr>
<th>Name</th>
<th>Format</th>
<th>Signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concatenation</td>
<td>• : :</td>
<td>( \bar{b}_v[n] \times \bar{b}_v[m] \rightarrow \bar{b}_v[n+m] )</td>
</tr>
<tr>
<td>Extraction</td>
<td>[j:i]</td>
<td>( \bar{b}_v[n] \rightarrow \bar{b}_v[j-i+1] ; j \geq i )</td>
</tr>
<tr>
<td>Addition</td>
<td>• + •</td>
<td>( \bar{b}_v[n] \times \bar{b}_v[n] \rightarrow \bar{b}_v[n] )</td>
</tr>
<tr>
<td>Subtraction</td>
<td>• − •</td>
<td>( \bar{b}_v[n] \times \bar{b}_v[n] \rightarrow \bar{b}_v[n] )</td>
</tr>
<tr>
<td>Unsigned shift to the right</td>
<td>( \gg • )</td>
<td>( \bar{b}_v[n] \rightarrow \bar{b}_v[n] )</td>
</tr>
<tr>
<td>Unsigned shift to the left</td>
<td>( \ll • )</td>
<td>( \bar{b}_v[n] \rightarrow \bar{b}_v[1] )</td>
</tr>
<tr>
<td>Reduction and</td>
<td>( \downarrow • )</td>
<td>( \bar{b}<em>v[n] = &amp;</em>{i=0}^{n-1} \bar{x}[i] ) (i.e., it is the “and” of all the bits in ( \bar{x} )).</td>
</tr>
</tbody>
</table>

QF_BV has two core operators:

- **Concatenation**
  \( \bar{x}[n] : \bar{y}[m] \) is a bit-vector of size \( n + m \) (\( \bar{x}[n+m] \)), such that \( \bar{x}[0] = \bar{y}[0] \) and \( \bar{x}[m+n-1] = \bar{x}[n-1] \). For example, \( 111 : 0 = 1110 \).

- **Extraction**
  \( x[j:i] \) is a bit-vector of size \( j - i + 1 \), extracted from the bit number \( i \) to the bit number \( j \), which can be defined based on the concatenation operator as \( x[j:i] = \vdash_{k=j}^{i} x[k] \). For example, \( 1100[j:0] = 100 \).

Moreover, we use arithmetic operators: *addition* (+) and *subtraction* (−), where the final carry bit is thrown away and the resulting bit-vector has the same size as the operands. Finally in the *unsigned shift to the right/left* the rightmost/leftmost bit is thrown away and *zero* is inserted from the left/right. For example, \( \gg 1100 = 0110 \) and \( \ll 1100 = 1000 \).

We also use bitwise operators like *negation* (!), *conjunction* (\&), *disjunction* (|), and *reduction and* (\( \downarrow \)). The *reduction and* operator is defined as \( \downarrow \bar{x}[n] = \&_{i=0}^{n-1} \bar{x}[i] \) (i.e., it is the “and” of all the bits in \( \bar{x} \)). The size of the resulting bit-vector is one. The bit
corresponds to the minimum value in \( \bar{x} \); in other words, it is equal to one if all the bits of the bit-vector \( \bar{x} \) are one, zero otherwise.

5.2 Bit-vector-based LTL Encoding Setup

Similarly to the classic Boolean encoding of Section 2.5, our encoding uses a bit-vector of size \( k + 2 \) to represent the truth values of each subformula of \( \phi \) from 0 to \( k + 1 \). However, we only introduce as many bit-vectors as the number of atomic propositions in the formula, and describe the values of the non-atomic subformulae as transformations on the former vectors. More precisely, for each \( p \in AP \), we introduce a bit-vector, \( \bar{p}_{[k+2]} \), such that \( \bar{p}_{[i]} \), with \( i \in [0, k + 1] \), captures the value of proposition \( p \) at instant \( i \). Recall that \( \bar{p}_{[0]} \) is the right-most (least significant) bit in \( \bar{p} \), and \( \bar{p}_{[k+1]} \) is the left-most (most significant) one.

In addition, we introduce a bit-vector, \( \bar{loop}_{[k+2]} (l) \), that contains (encoded in binary) the position of the loop in interval \([0, k + 1]\) (the position of the first state \( s \) in \( \alpha s \beta s \)).

For the sake of uniformity in using Bit-Vector Logic operators to capture the semantics of LTL formulae, we encode \( \bot \) (false) as \( \bar{0}_{[k+2]} \) (i.e., a sequence of zeros) and \( \top \) (true) as \( \bar{1}_{[k+2]} \) (i.e., a sequence of ones), so the size of all bit-vectors used in the encoding is \( k + 2 \).

5.3 Bit-vector-based Encoding

To introduce the bit-vector-based encoding of LTL formulae, it is useful to first define some auxiliary operators on bit-vectors that will be exploited in the following. Table 5.2 defines these auxiliary operators.

### Table 5.2: Definition of auxiliary operators.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rev ( \bar{x} )</td>
<td>( \bar{x} \rightarrow \bar{x} )</td>
</tr>
<tr>
<td>( \bar{x} \cup_{nl} \bar{y} )</td>
<td>( \bar{y} (\bar{x} \cup \neg \text{Rev} (\text{Rev} (\bar{x} \cup \bar{y}) + \text{Rev} \bar{y})) )</td>
</tr>
</tbody>
</table>

\( \text{Rev} \bar{x} \) reverses the order of the bits in bit-vector \( \bar{x} \). \( \bar{x} \cup_{nl} \bar{y} \) produces a bit-vector, say \( \bar{z} \), such that, if one takes \( \bar{x} \) and \( \bar{y} \) to represent the values of some formulae \( x, y \) in \([0, k + 1]\), then \( \bar{z} \) corresponds to the value of \( x \cup_{nl} y \) when one considers only finite models (i.e., there is no loop, hence the subscript \( nl \)). Later in this section, we give an example of computation of \( \bar{x} \cup_{nl} \bar{y} \).

In the bit-vector-based encoding of LTL, the bit-vector capturing the value of a formula \( \phi \) in \([0, k + 1]\) is obtained by recursively performing operations on the bit-vectors corresponding to the subformulae of \( \phi \). The operations performed depend on the structure of \( \phi \). Table 5.3 shows the case in which the main connective in \( \phi \) is a Boolean one. For example, if \( \phi \) is of the form \( \neg \psi \), then its corresponding bit-vector is obtained by applying bit-wise negation to the bit-vector, \( \bar{\psi} \), corresponding to subformula \( \psi \). Similarly for the other cases. Table 5.4 shows the transformations in the case of both
Chapter 5. Bit-vector-based Encoding for LTL and CLTL

Table 5.3: QF_BV encoding of propositional formulae.

<table>
<thead>
<tr>
<th>( \phi )</th>
<th>bit-vector encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \neg \psi )</td>
<td>( \neg \psi )</td>
</tr>
<tr>
<td>( \psi_1 \land \psi_2 )</td>
<td>( \psi_1 \land \psi_2 )</td>
</tr>
<tr>
<td>( \psi_1 \lor \psi_2 )</td>
<td>( \psi_1 \lor \psi_2 )</td>
</tr>
</tbody>
</table>

future (X, U) and past (Y, S) temporal operators.

Table 5.4: QF_BV encoding of basic temporal operators.

<table>
<thead>
<tr>
<th>( \phi )</th>
<th>bit-vector encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>( X\psi )</td>
<td>( \psi[i+1] \ldots \psi[k+1:1] )</td>
</tr>
<tr>
<td>( \psi_1 U \psi_2 )</td>
<td>( \psi_1 \psi_2 \oplus ((\psi_1 U \psi_2)[i] : \psi_2) )</td>
</tr>
<tr>
<td>( Y\psi )</td>
<td>( \ll \psi )</td>
</tr>
<tr>
<td>( \psi_1 S \psi_2 )</td>
<td>( \ll \psi_2</td>
</tr>
</tbody>
</table>

We illustrate the cases for the past operators first, which are a bit simpler, and then we focus on the future ones.

Yesterday – Given the semantics of formula \( Y\psi \), where \( Y\psi \) holds at \( i \) iff \( \psi \) holds at \( i-1 \), the bit-vector for \( Y\psi \) is the one for \( \psi \), but shifted “to the left” (from \( i-1 \) to \( i \), recall that position 0 in bit-vectors is the rightmost one). Note that, by definition of shifting to the left, the rightmost bit of \( \ll \psi \) is 0, which is consistent with the semantics of \( Y\psi \) in the origin.

Table 5.5 shows an example of calculation of the bit-vector for \( \psi_1 S \psi_2 \), given bit-vectors \( \psi_1 \) and \( \psi_2 \).

Since – First of all, recall that, informally, \( \psi_1 S \psi_2 \) holds at \( i \) iff either \( \psi_2 \) holds at \( i \), or \( \psi_1 \) holds at \( i-1 \), ... until an instant \( i' < i \) in which \( \psi_2 \) holds (\( \psi_1 \) can hold in \( i' \) or not). Said in another way, if \( \psi_2 \) holds in \( i' \), then \( \psi_1 S \psi_2 \) holds there, and in all instants \( i'+1, i'+2, \ldots \) in which \( \psi_1 \) holds consecutively. We use addition to capture this mechanism through which the truth of \( S \) “carries” to the left from when \( \psi_2 \) holds, as long as \( \psi_1 \) holds. Consider term \( \psi_1 \ll \psi_2 + \psi_2 \); whenever \( \psi_2[i] \) is 1, this generates a carry (since both bits are 1), which propagates as long as \( \psi_1[i] \) is 1, as between bits 1-2 and 6-8 in Table 5.5. The net effect is that in the sum the bits from \( i' \) until \( \psi_1 \) stops holding are set to 0, and the others are set to 1 (we do not delve into the details of some special cases, which are covered by the correctness proof of Section 5.4). Starting from this basis, the rest of the operations are necessary to set to 1 exactly all the bits in which \( S \) holds (notice that, in this case, \( \psi_1 S \psi_2 \) holds not only where \( \psi_1 \ll \psi_2 + \psi_2 \) is 0, but also at position 3, where \( \psi_2 \) holds). More precisely, the result of \( \psi_1 \ll \psi_2 + \psi_2 \) is
5.3. Bit-vector-based Encoding

Table 5.5: An example of calculation of bit-vector for $S$.  

<table>
<thead>
<tr>
<th>bit-vector</th>
<th>11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\leftarrow \psi_1$</td>
<td>1 1 0 1 1 1 1 0 0 1 0 1</td>
</tr>
<tr>
<td>$\leftarrow \psi_2$</td>
<td>0 0 0 0 0 1 0 0 1 0 1 0</td>
</tr>
<tr>
<td>$\leftarrow \psi_1</td>
<td>\psi_2$</td>
</tr>
<tr>
<td>$\overrightarrow{bv}_1 ( \overrightarrow{\psi}_1 , , \overrightarrow{\psi}_2 ) + \overrightarrow{\psi}_2$</td>
<td>1 1 1 0 0 0 1 1 0 0 0 1 1</td>
</tr>
<tr>
<td>$\overrightarrow{bv}_2$</td>
<td>0 0 0 1 1 1 0 0 0 1 1 0</td>
</tr>
<tr>
<td>$\overrightarrow{bv}_3$</td>
<td>$\overrightarrow{bv}_2$ &amp; $\overrightarrow{\psi}_1$</td>
</tr>
<tr>
<td>$\overrightarrow{\psi}_1 S \overrightarrow{\psi}_2$</td>
<td>$\overrightarrow{bv}_3$ &amp; $\overrightarrow{\psi}_2$</td>
</tr>
</tbody>
</table>

bit-wise complemented to bring the 0s to 1s; it is then filtered against $\leftarrow \psi_1$ with a bitwise “and” to eliminate those cases in which the result of the sum is 0 because both $\leftarrow \psi[i]$ and $\leftarrow \psi_2[i]$ are 0 and there is no carry from $i - 1$ ($\leftarrow \psi_1 S \leftarrow \psi_2$ does not hold there); finally, the bit-wise “or” with $\leftarrow \psi_2$ sets to 1 all those positions in which $\leftarrow \psi_2$ holds, since $\leftarrow \psi_1 S \leftarrow \psi_2$ holds there. Figure 5.1 gives hints about how the nested formula $(\leftarrow \psi_1 S \leftarrow \psi_2) SY \phi_3$ is calculated assuming the input subformulae are initialized and $K$ is 2.

We now illustrate the encoding of the future operators $X$, $U$. **Next** – The encoding of formula $X \leftarrow \psi$ is essentially dual to that of $Y \leftarrow \psi$, i.e., a bit-wise shift to the right of $\leftarrow \psi$. In fact, $X \leftarrow \psi$ holds at $i$ iff $\leftarrow \psi$ holds at $i + 1$. However, a true right-shift would always introduce a 0 at position $k+1$, which would be incorrect. In fact, recall that, in bounded encodings such as the classic one presented in Section 2.5, we need the state to repeat at positions $k+1$ and $l$, with the latter corresponding to the position of the first $s$ in $\alpha_s \beta_s$, which in the bit-vector encoding is represented by the value of bit-vector $\overrightarrow{\text{loop}}$. Hence, the value of $X \leftarrow \psi$ at positions $k+1$ and $l$ must be the same (this is also true for $Y \leftarrow \psi$, but it is achieved in a different way than for $X \leftarrow \psi$, as it will be explained later). Note that the value of $X \leftarrow \psi$ at position $l$ is the same as $\leftarrow \psi$ at position $l + 1$, hence, the bit-vector corresponding to $X \leftarrow \psi$ is obtained by concatenating $\leftarrow \psi[l+1]$ with $\leftarrow \psi[k+1:1]$.

**Until** – The bit-vector corresponding to $\leftarrow \psi_1 U \leftarrow \psi_2$ is computed by exploiting the operator $U_{nl}$ introduced above. In fact, the operations performed by $U_{nl}$ are the same as those for the computation of $\leftarrow \psi_1 S \leftarrow \psi_2$, but carried out left-to-right instead of right-to-left. To achieve this, bit-vectors $\overrightarrow{x}$ | $\overrightarrow{y}$ and $\overrightarrow{y}$ are reversed through $\text{Rev}$ before being added together, and the result is also reversed.

As mentioned above, the operations performed by operator $U_{nl}$ are the same as those for the encoding of $\leftarrow \psi_1 S \leftarrow \psi_2$, but carried out in the reverse direction. Hence, they produce, from bit-vectors $\overrightarrow{x}$ and $\overrightarrow{y}$, a bit-vector that corresponds to the truth of $xUy$ evaluated over finite models, whereby $xUy$ holds at position $k+1$ iff $y$ holds at $k+1$. However, when finite traces of the form $\alpha_s \beta_s$ are used to represent ultimately periodic models, one must take into account that, in the encoding, $\leftarrow \psi_1 U \leftarrow \psi_2$ can hold in $k+1$ (i.e., state $s$) also if $\leftarrow \psi_1$ holds there, and $\leftarrow \psi_2$ holds somewhere in $\beta$ (with $\leftarrow \psi_1$ holding in the prefix...
of $\beta$ until then). Since the states in positions $k + 1$ and $l$ are the same (they are both $s$), this is the same as saying that, in position $l$, $\psi_1 U \psi_2$ holds in the finite model, that is, the $l$-th bit of $\psi_1 U_{nl} \psi_2$ is 1. If, conversely, the $l$-th bit of $\psi_1 U_{nl} \psi_2$ is 0, then there is no point in $\beta$ in which $\psi_2$ holds, with $\psi_1$ holding until then, so $\psi_1 U \psi_2$ does not hold in state $s$ (at position $l$ or at position $k + 1$). Finally, to correctly compute the bit-vector corresponding to $\psi_1 U \psi_2$, we compute $\psi_1 U_{nl} \psi_2$, take its $l$-th bit to determine whether $\psi_1 U \psi_2$ holds in $s$, then use bit-vector $\psi_1 U_{nl} \psi_2[k]$ as the second argument of $U_{nl}$; in fact, as mentioned above, by definition the value of $\psi_1 U_{nl} \psi_2[k]$ is $1$ if $\psi_2[k+1] = 1$.

Whereas the functions computing bit-vectors for future operators $X, U$ by construction impose that the value of the subformula in positions $l$ and $k + 1$ is the same, the same does not happen for the functions that compute the bit-vectors for past operators $Y, S$. These so-called "last state constraints" (see also Section 2.5) are easily included in
the encoding by adding, for each formula $Y\psi$, the constraint $(\llcorner \psi \lrcorner)[l] = (\llcorner \psi \lrcorner)[k+1]$, and similarly, *mutatis mutandis*, for each formula $\psi_1 S \psi_2$. The “last state constraints” must be added for all subformulas, including propositional letters, so for each $p \in AP$ we also include the constraint $\llcorner p \lrcorner[l] = \llcorner p \lrcorner[k]$. Notice that it is not necessary to include the “last state constraints” for each subformula of the form $\neg \psi$, $\psi_1 \land \psi_2$ and $\psi_1 \lor \psi_2$, as they are automatically guaranteed recursively. We indicate this set of constraints as $|BVLastStateConstraints|_k$.

Finally, the so-called “loop constraints” (see Section 2.5) are easily imposed by adding, for each $p \in AP$, the constraint $\llcorner p \lrcorner[l-1] = \llcorner p \lrcorner[k]$. We name this set of constraints $|BVLoopConstraints|_k$.

Then, given an LTL formula $\phi$, the complete bit-vector-based encoding, called $\phi_{bv}$, is given by:

- the set $|BVLastStateConstraints|_k$, which includes constraints for each $p \in AP$, and for each past formula $Y\psi$ and $\psi_1 S \psi_2$;
- the set $|BVLoopConstraints|_k$, which includes a constraint for each $p \in AP$;
- constraint $\llcorner \phi \lrcorner[0] = 1$, where $\llcorner \phi \lrcorner$ is the bit-vector obtained through the transformations above.

For example, consider formula $\neg X p \lor (q U Y p)$. Its complete encoding is given by the following formula:

$$
\llcorner p \lrcorner[l] = \llcorner p \lrcorner[k+1] \land \llcorner q \lrcorner[l] = \llcorner q \lrcorner[k+1] \land (\llcorner p \lrcorner)[l] = (\llcorner p \lrcorner)[k+1] \land \\
\llcorner p \lrcorner[l-1] = \llcorner p \lrcorner[k] \land \llcorner q \lrcorner[l-1] = \llcorner q \lrcorner[k] \land \\
\left( \neg (\llcorner p \lrcorner[l+1] \vdots \llcorner p \lrcorner[k+1:1]) \mid (\llcorner q \lrcorner U_{nl}(\llcorner q \lrcorner U_{nl}(\llcorner p \lrcorner))[l] \vdots (\llcorner p \lrcorner)[k:0]) \right)[0] = 1
$$

It is customary to define the other temporal operators from the basic ones as abbreviations. In some cases we can use the abbreviation to further simplify the encoding of these operators by exploiting the properties of bit-vector operations. First of all, we can introduce operators $\llcorner x \lrcorner R_{nl} \llcorner y \lrcorner$ and $F_{nl} \llcorner y \lrcorner$, shown below, whose definitions correspond, respectively, to the simplified versions of $\neg(\neg \llcorner x \lrcorner U_{nl} \neg \llcorner y \lrcorner)$ and $\top U_{nl} \llcorner y \lrcorner$.

**Table 5.6: Definition of additional auxiliary operators.**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\llcorner x \lrcorner R_{nl} \llcorner y \lrcorner$</td>
<td>$\llcorner y \lrcorner &amp; (\llcorner x \lrcorner</td>
</tr>
<tr>
<td>$F_{nl} \llcorner y \lrcorner$</td>
<td>$\neg(\llcorner y \lrcorner &amp; \text{Rev}(\llcorner y \lrcorner - \top))$</td>
</tr>
</tbody>
</table>

Using these operators, we can in turn simplify the encoding of derived temporal operators $T$ and $P$.

Finally, the encoding of “always $\psi$” (written $W$), which is defined as $W\psi = G\psi \land H\psi$ can be simplified as shown above by considering that its value is 1 throughout $[0, k + 1]$ if $\llcorner \psi \lrcorner$ has value 1 everywhere, otherwise its value is 0 at every position.
Chapter 5. Bit-vector-based Encoding for LTL and CLTL

Table 5.7: \(QF_{BV}\) encoding of derived temporal operators.

<table>
<thead>
<tr>
<th>(\phi)</th>
<th>bit-vector encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\psi_1 R \psi_2)</td>
<td>(\psi_1 R_{nl}((\psi_1 R_{nl} \psi_2)[0] :: \psi_2[0:k]))</td>
</tr>
<tr>
<td>(F \psi)</td>
<td>(F_{nl}(\psi)[0] :: \psi[0:k]))</td>
</tr>
<tr>
<td>(\psi_1 T \psi_2)</td>
<td>(\psi_2 \land (\psi_1</td>
</tr>
<tr>
<td>(P \psi)</td>
<td>(\psi</td>
</tr>
<tr>
<td>(W \psi)</td>
<td>(::k+2 \psi)</td>
</tr>
</tbody>
</table>

We conclude this section by remarking that if one wants to consider formulae in PNF (see Section 2.4), which entails that operator \(Z\) be introduced, the semantics of \(Z \psi\) is simply captured by the transformation \(\ll \psi | 1\).

5.4 Correctness Proof

In this section, we show that the new, bit-vector-based bounded encoding of LTL formulae is equivalent to the classic Boolean one introduced in Section 2.4.

To show the equivalence, it is natural to consider a bit-vector \(\ll x\) of size \(n\), whose bits are \(\ll x[0] \ldots, \ll x[n-1]\), as a set of \(n\) Boolean variables \([[x]]_0, \ldots, [[x]]_{n-1}\). An operation on bit-vectors (e.g., \(\ll\)) returns a bit-vector, which corresponds to its own set of Boolean propositions; hence, for example, bit-vector \(\ll \ll x\), obtained by shifting \(\ll x\) to the left, comprises \(n\) bits, \((\ll \ll x)_0, \ldots, (\ll \ll x)_{n-1}\), which in turn correspond to Boolean variables \([[\ll \ll x]]_0, \ldots, [[\ll \ll x]]_{n-1}\).

Recall that, given an LTL formula \(\phi\), we indicate by \(\phi_B\) the set of formulae that correspond to the Boolean encoding of \(\phi\), and by \(\phi_{bv}\) the ones of the bit-vector-based encoding of \(\phi\) (see Sections 2.4 and 5.3 for the definitions of \(\phi_B\) and \(\phi_{bv}\)). We have the following result.

Given an LTL formula \(\phi\), the encoding \(\phi_B\) is equivalent to the encoding \(\phi_{bv}\).

We prove the equivalence by showing that every constraint in \(\phi_B\) corresponds to a constraint in \(\phi_{bv}\) and vice-versa. First of all, we remark that the bit-vector encoding assumes the existence of a loop starting at position \(l\), so we focus on this case for the proof. This is without loss of generality, as it is always possible to extend an aperiodic trace with a “dummy” loop at the end, in which nothing happens. Notice also that, since \(l\) is the position of the first \(s\) in \(\alpha s\beta s\), we have that \(0 \leq l < k + 1\).

Let us first consider the \(\left|LoopConstraints\right|_k\). It is easy to see that they directly correspond to \(\left|BVLoopConstraints\right|_k\), which in fact impose that, for each \(p \in AP\), in the corresponding bit-vector \(\ll p\) it holds that \(\ll p[l-1] = \ll p[k]\). Given the correspondence between bits of bit-vectors and propositional letters introduced in the Boolean encoding, this is the same as saying, for each \(p \in AP\), that \([[p]]_{l-1} \iff [[p]]_k\).

We will tackle \(\left|LastStateConstraints\right|_k\) at the end of the proof. We now focus on the encoding of propositional connectives, i.e. \(\left|PropConstraints\right|_k\). The bit-vector of
formula $\psi_1 \land \psi_2$ is simply $\psi' \land \psi''$, that is, for each bit $i \in [0, k]$ (we will deal with the $k + 1$-th bit in the $|LastStateConstraints|_k$), it is $(\psi' \land \psi'')^i = 1$ iff $\psi'_i = 1 \land \psi''_i = 1$, which corresponds to Boolean constraints $[|\psi_1 \land \psi_2|]_i \leftrightarrow [||\psi_1||_i \land ||\psi_2||_i]$, i.e., $[PropConstraints]_i$, for the $\land$ connective. Similarly for the $\lor$ connective. As far as the $\neg$ connective is concerned, recall that the Boolean encoding assumes formulae in Positive Normal Form (PNF), an optimization for saving intermediate Boolean variables which we do not pursue in the bit-vector-based encoding. Even in the Boolean encoding, the optimization could be eliminated without impacting on the correctness of the encoding, by introducing a Boolean variable $[\neg \psi]$ for each formula of the form $\neg \psi$, with the constraint that $[\neg \psi]_i \leftrightarrow \neg[\psi]_i$. Then, it is obvious that such a constraint corresponds to the semantics of the bit-wise negation: $(\neg \psi^i)[i] = 1$ iff $\psi^i[i] = 0$.

Let us now focus on $|TempConstraints|_k$, starting from those regarding past operators $\mathit{Y}$ and $\mathit{S}$. The semantics of $\mathit{Y}\psi$ is captured by the transformation $\ll \psi$, which by definition means that, for all $i \in [1, k + 1]$, it holds that $(\ll \psi)[i] = \psi[i−1]$. This corresponds to the constraint that, for $1 \leq i \leq k + 1$, $[\mathit{Y}\psi]_i \leftrightarrow [\psi]_{i−1}$. In addition, by definition of $\ll$, $(\ll \psi)^[0] = 0$, which corresponds to the constraint $\neg[\mathit{Y}\psi]_0$ that appears in the Boolean encoding. The encoding of $\psi_1 \mathit{S}\psi_2$ is trickier. In the Boolean encoding, we have that, for $i \in [1, k + 1]$, $[\psi_1 \mathit{S}\psi_2]_i \leftrightarrow [||\psi_1||_i \lor (||\psi_1||_1 \land ||\psi_2||_{i−1})]$, with the additional constraint that $[\psi_1 \mathit{S}\psi_2]_0 \leftrightarrow [||\psi_1||_0]$. To show how this is equivalent to the bit-vector encoding, in which the bit-vector representing $\psi_1 \mathit{S}\psi_2$ is $\psi'_2((\psi'_1 \mathit{S}\psi'_2) \lor \psi'_2)$, we need to show that, for $i \in [1, k + 1]$, if $\psi'_2[i] = 0$ and $\psi'_1[i] = 1$, then $((\psi'_1 \mathit{S}\psi'_2) \lor \psi'_2)[i] = 1$ iff $\psi_1 \mathit{S}\psi_2$ holds in $i − 1$ (if $\psi'_2[i] = 1$, i.e., $[||\psi_2||_i]$ is true, or $\psi'_1[i] = 0$, i.e., $[||\psi_1||_i]$ is false, the Boolean and bit-vector encodings clearly yield the same result). Then, we need to determine when $((\psi'_1 \mathit{S}\psi'_2) \lor \psi'_2)[i] = 0$, provided $\psi'_2[i] = 0$ and $\psi'_1[i] = 1$. Notice that, if $\psi'_2[i] = 0$ and $\psi'_1[i] = 1$, then $((\psi'_1 \mathit{S}\psi'_2) \lor \psi'_2)[i] = 0$ iff there is a carry from $i − 1$. This occurs, recursively, iff either $\psi'_2[i−1] = 0$, $\psi'_1[i−1] = 1$ and there is a carry from $i − 2$, or $\psi'_2[i−1] = 1$.

By inductive reasoning, for $((\psi'_1 \mathit{S}\psi'_2) \lor \psi'_2)[i]$ to be 0, there must be $i' < i$ in which $\psi'_2[i'] = 1$, and for all $i < i'' < i'$ it is $\psi'_2[i''] = 0$ and $\psi'_1[i''] = 1$. In other words, $\psi_1 \mathit{S}\psi_2$ holds in $i − 1$. If, on the other hand, $\psi_1 \mathit{S}\psi_2$ holds in $i$, there must be $i' < i$ where $\psi'_2[i'] = 1$ and for all $i < i'' < i'$ it is $\psi'_2[i''] = 1$. In this case in $i'$ $((\psi'_1 \mathit{S}\psi'_2) \lor \psi'_2)[i']$ generates a carry, which propagates to the left, until in $i$ it produces a 0. Notice that, since in position 0 there can be no carry, $((\psi'_1 \mathit{S}\psi'_2) \lor \psi'_2)[0] = 0$ iff $\psi'_2[0] = 1$, which corresponds to the Boolean encoding of $\psi_1 \mathit{S}\psi_2$ in 0.

Let us now focus on the future operators $\mathit{X}$ and $\mathit{U}$. Since the bit-vector for $\mathit{X}\psi$ is $\psi'[i+1] :: \psi'[k+1]$ for $i \in [0, k]$ (we will deal with the case $i = k + 1$ when focusing on $|LastStateConstraints|_k$), we have that $(\psi'[k+1] :: \psi'[k+1]:[i] = \psi'_{i+1}[i]$, which corresponds to the Boolean encoding of $\mathit{X}\psi$, i.e., $[\mathit{X}\psi]_i \leftrightarrow [||\psi||_{i+1}]$. The encoding of $\mathit{U}$ relies on operator $\mathit{U}_{nl}$, whose definition coincides with that of $\mathit{S}$, except that the arguments are reversed before applying the sum, and reversed again after having applied it. As a consequence, the properties of $\mathit{U}_{nl}$ are the same as those of the encoding of $\mathit{S}$, except that the bit vectors are considered in reverse order. Then, we can conclude that, for $i \in [0, k]$, $((\mathit{U}_{nl} \mathit{U}_{nl}) y)[i] = 1$ iff there is $i < i' < k + 1$ such that $\psi'[i'] = 1$. and,
for all \( i \leq i'' < i' \), \( \sum_{i''}^{i'} = 1 \) holds. In addition, \((\oplus \mathbf{U}_{nl} \chi)_{[k+1]} = 1 \) iff \( \chi_{[k+1]} = 1 \).

The Boolean encoding of \( \mathbf{U} \) differs from that of \( \mathbf{S} \) because, unlike the latter, where the value of \( \psi_1 \mathbf{S} \psi_2 \) in 0 depends exclusively on the value of \( \psi_2 \) there, in the former the value in \( k \) depends on whether there is a position in the loop (i.e., in the part \( s \beta \) of the trace) where \( \psi_2 \) holds, as defined by constraints \( \text{[Eventualities]}_k \). Given the properties of operator \( \mathbf{U}_{nl} \), and in particular the fact that \((\oplus \mathbf{U}_{nl} \chi)_{[k+1]} = 1 \) iff \( \chi_{[k+1]} = 1 \), the value of \((\oplus \psi_1 \mathbf{U}_{nl} (\psi_1 \mathbf{U}_{nl} \psi_2)_{[l]} :: \psi_2)_{[2]} = 1 \) is iff either \( \psi_2 = 1 \) or \( \psi_1 = 1 \) and \((\psi_1 \mathbf{U}_{nl} \psi_2)_{[l]} = 1 \).

Both cases must be in a position in the loop in which \( \psi_2 \) holds; the first is evident (that position is \( k \)); the second derives from the fact that by the properties of \( \mathbf{U}_{nl} \), \( \psi_1 \mathbf{U}_{nl} \psi_2 \) is 1 in \( l \) iff there is \( l \leq i' \leq k \) in which \( \psi_2 \) holds (notice that, as it will be shown below, it is guaranteed that \( \psi_2_{[l]} = \psi_2_{[k+1]} \)).

Finally, we need to show that the \( \text{[LastStateConstraints]}_k \) are also captured by the bit-vector-based encoding. To this end, recall that for propositional letters and past operators \( \mathbf{Y} \) and \( \mathbf{S} \) the following constraints are explicitly introduced: \( \frac{\chi_{[l]}}{\chi_{[l]}} = \langle \langle \chi_{[l]} \rangle \rangle \land (\psi_1 \mathbf{U}_{nl} (\psi_1 \mathbf{U}_{nl} \psi_2)_{[l]} :: \psi_2)_{[2]} = 1 \) if \( (\psi_1 \mathbf{U}_{nl} \psi_2)_{[l]} = 1 \), then there is \( l \leq i' \leq k + 1 \) such that \( \psi_2 = 1 \) and \( \psi_1 = 1 \) for all \( l \leq i'' < i' \). If \( i' < k + 1 \) it is easy to see that \((\psi_1 \mathbf{U}_{nl} (\psi_1 \mathbf{U}_{nl} \psi_2)_{[l]} :: \psi_2)_{[k+1]} = 1 \). If, instead, \( i' = k + 1 \), then \((\psi_1 \mathbf{U}_{nl} \psi_2)_{[l]} = \psi_2_{[k+1]} \), so \((\psi_1 \mathbf{U}_{nl} \psi_2)_{[l]} = \psi_2_{[k+1]} \), if \((\psi_1 \mathbf{U}_{nl} \psi_2)_{[l]} = 0 \), then there is no \( l \leq i' \leq k + 1 \) such that \( \psi_2 = 1 \) and \( \psi_1 = 1 \) for all \( l \leq i'' < i' \). Then, \( \text{a fortiori} \) also \((\psi_1 \mathbf{U}_{nl} (\psi_1 \mathbf{U}_{nl} \psi_2)_{[l]} :: \psi_2)_{[k+1]} = 1 \). Finally, the fact that \( \text{[LastStateConstraints]}_k \) hold also for formulae of the form \( \neg \psi, \psi_1 \wedge \psi_2 \) and \( \psi_1 \lor \psi_2 \) can be easily shown by induction, where the base cases are those already tackled above.

To conclude the proof, we remark that the classic Boolean encoding natively defines the semantics of operators \( \mathbf{R} \) and \( \mathbf{T} \). In our bit-vector-based encoding, instead, the encoding of these operators exploits their definition as abbreviations for formulae involving \( \mathbf{U} \) and \( \mathbf{S} \); hence, the correctness of the encoding in this case derives directly from that of the encoding of formulæ \( \neg \psi, \psi_1 \wedge \psi_2 \) and \( \psi_1 \lor \psi_2 \).

### 5.5 Combining Bit-vectors and Arithmetic to solve CLTL

A CLTL formula (introduced in Section 2.6) can be considered as an LTL formula entailing arithmetic constraints, as a layer on top of temporal layer. To separate the
5.5. Combining Bit-vectors and Arithmetic to solve CLTL

Arithmetic layer from the temporal one, each arithmetic constraint is expressed through linear integer/real arithmetic logic formulae at each time instant; each constraint is then replaced by a fresh atomic proposition (essentially, a Boolean abstraction of the arithmetic constraint), which acts as placeholder in the temporal formula. As a result, we obtain a temporal logic formula that is free from any arithmetic constraints, which is conjoined with the assertions capturing these constraints in the corresponding logic (i.e., the concrete representation of the constraints). For example, consider formula \((X.x > x)S(v = 1)\) — where \(x\) and \(v\) are integer-valued, time-dependent variables — that states that the value of \(x\) strictly increased since an instant in the past when the value of \(v\) was equal to 1. To encode it, according to [10] we introduce two sets of integer variables: \(k + 2\) integer variables \(v_0, \ldots v_{k+1}\), which capture the value of \(v\) at every time instant in \([0..k + 1]\), and \(k + 3\) integer variables \(x_0, \ldots x_{k+2}\), which capture the value of \(x\) and \(X x\) at each position in \([0..k + 1]\) (for example, the value of \(X x\) at position \(k + 1\) is given by \(x_{k+2}\)). These variables are used to impose the constraints that are necessary to capture the semantics of arithmetic variables/clocks in CLTL and CLTLoc, as defined in [10, 12, 14]. For example, for clocks in CLTLoc, we need to introduce constraints that state that all clocks advance of the same quantity, unless they are reset.

In addition, we introduce two bit-vectors, \(bv_{X x > x}\) and \(bv_{v = 1}\), which represent the value of the corresponding atomic formulae in \([0..k + 1]\). Then, \(\bigwedge_{i=0}^{k+1} (bv_{X x > x} \iff x_{i+1} > x_i)\) and \(\bigwedge_{i=0}^{k+1} (bv_{v = 1} \iff v_i = 1)\) are asserted, and the value of formula \((X x > x)S(v = 1)\) is given by bit-vector \(bv_{v = 1} \mid (bv_{X x > x} \& !((bv_{X x > x}|bv_{v = 1}) + bv_{v = 1}))\), as defined in Section 5.3.
CHAPTER 6

Tools and Experimental Evaluation

To fulfill the goals of this study, we have implemented, tested, optimized, and finally verified the research outcome, step by step. We also compared the works and tools of ours with other works and corresponding tools. Section 6.1 introduces Corretto, our verification tool, where our flexible modular formalization of UML/eFT-UML models is implemented. To reason about different semantics of a model, we investigated a relatively small SD to be both easily understandable and containing various CFs and different behaviors with respect to the decisions made for the semantic variation points in Section 6.2.

Since the efficiency of our verification tool highly depends on its main verification engine, that is Zot, we proposed novel encodings that highly influences its efficiency. Section 6.3 introduces Zot, our Bounded Model/Satisfiability Checker, on top of which our bit-vector-based encodings for both LTL and CLTL formulae are implemented. We performed scalability analysis for various LTL encodings and compared bvzot the Zot plugin, where our LTL encoding is implemented, with the state of the art model checker NuSMV [23], that contains implementation of several LTL encodings. Section 6.4 reports the experimental results and reason about why our tool scales better than the other tools. Finally, Section 6.5 compares ae2bvzot where our CLTL encoding is implemented, to the only existing bounded encoding of CLTL that is another plugin of Zot, namely ae2zot.

6.1 Corretto

Corretto\(^1\) is our eFT-UML verification tool, built on top of several interoperable tools. The goal of Corretto is to hide formalization and verification details from designers who

\(^1\)https://github.com/deib-polimi/Corretto
might not be expert in formal method, and let them focus on model behavior rather than formal details. Designers can simulate the behavior of the diverse semantics and verify the satisfaction of the properties of interest. They can thus understand how the system behaves, or work on the satisfaction of the properties and then obtain the guarantees the system must offer. Corretto aims to provide these features while users neither need to leave the UML mindset nor interpret the verification results produced by underlying tools. The formal verification of eFT-UML models is divided in the four phases. Corretto supports these phases using both well established and prototype tools:

**Modeling:** During this phase the user builds the model and specifies the property of interest using eFT-UML. Corretto exploits Papyrus\(^2\), which provides an integrated and user-oriented environment for editing models created with UML and related modeling languages such as SysML [47] and MARTE [66]. Papyrus is integrated with the Eclipse Modeling Framework. After the installation, Papyrus is enriched with all the eFT-UML stereotypes to build the model and specify the properties of interest.

**Transformation:** eFT-UML model is translated —together with the properties to be verified— into its formal representation during this phase. The result, that is, the formal model, remains hidden to the user. To this end UML2ZOT, a Java component, is created that takes the eFT-UML model produced by Papyrus in XMI format and automatically translates it into a single TRIO formula. UML2ZOT is built on top of the Eclipse UML2 library, the Eclipse standard to read, create and modify UML compliant models.

**Verification:** During this phase the user-defined property is checked against the formal model. Corretto uses Zot [71], a bounded model/satisfiability checker that checks satisfiability of TRIO formulae by translating it into other decidable logics. Satisfiability of the translated formula, that is a combination of several logics, is in turn checked by a SAT or SMT (Satisfiability Modulo Theories) solver (e.g., Z3 [64]). If the property specified by the user holds, then s/he is just notified of the result. If the property does not hold, Zot returns a textual counterexample, that is a trace of the system violating the property.

**Traceability:** During this phase the result produced by Zot is inspected using a high-level representation. The aim is to bring verification results back to the modeling level the user is familiar with. If the property does not hold, the Traceability Tool visualizes the textual counterexample produced by Zot within Papyrus. The goal of this simple tool is to shorten the gap between the verification domain, to which the Zot textual counterexample belongs, and the modeling domain that is familiar for the user. Currently, this is achieved by simply linking the counterexample elements to their corresponding UML elements, but more sophisticated solutions could be built on top of this simple bridge.

### 6.2 Experimental Evaluation for Formal SDs

The proposed modular semantics for SDs has been implemented within Corretto. We used the resulting toolset for verifying the SD of Figure 6.1. For each module, the following properties were specified:

\(^2\)eclipse.org/papyrus
6.2. Experimental Evaluation for Formal SDs

Keywords, it pings server1 and server2 in parallel and passes the keywords to the first server that replies. app also periodically updates the screen with the results obtained from the server.

We compared the different configurations and present some of their discrepancies, in three ways:

(i) We injected the traces $T_1$ and $T_2$ introduced below, as additional specifications for the system and checked whether the system were satisfiable.

(ii) We asserted properties $P_1$ and $P_2$, defined below, to check whether they held for the system.

(iii) We imposed the time constraints introduced below on the system to study some real-time requirements.

By checking the incomplete trace $T_1$, we are trying to find out whether or not it is possible for replyS1 to be sent before pingS2 is received:\footnote{The notation we use decomposes a message in the two events that correspond to sending (!) and receiving (?). The shortcut !?m is equivalent to !m and ?m at the same time instant, and a pair of ⟨⟩ enclose the events that happen at the same time.}

\[
T_1 = \{⟨!?pingS1, pingS2⟩, ⟨replyS1⟩, ⟨!?pingS2⟩\}
\]

We check if the occurrences of events in different iteration can interleave, by checking the incomplete trace $T_2$:

\[
T_2 = \{⟨!?searchNextS1_{It1}⟩, ⟨!?resultS1_{It1}⟩, ⟨updateResultS1_{It1}⟩, ⟨!?searchNextS1_{It2}⟩, ⟨!?resultS1_{It2}⟩, ⟨updateResultS1_{It1}⟩, ⟨updateResultS1_{It2}⟩\}, \text{ where } ItX \text{ means the } x^{th} \text{ iteration, is}
\]

![SDSearch Sequence Diagram](image-url)
questionable.

Table 6.1 summarizes the results produced by the tool. Zot allows one to inject a partial (or complete) trace $T$ and check whether it complies with the model. If it does, Zot returns a complete version of $T$ (SAT) as output, otherwise the output is empty (UNSAT). We used this feature to analyze traces $T_1$ and $T_2$ and confirm the informal results already introduced in Section 4.2: $T_1$ is not satisfiable for those configurations that adopt the synchronous composition of combined fragments, whereas $T_2$ is satisfiable only for Config5 and Config6, that is, for those configurations that exploit weak sequencing for composing fragments.

Property $P_1$ is defined as follows: “If server1 sends a reply to app one time instant before server2 does, app will not send any search request to server2 during the current execution of diagram SDSearch”. Its formal equivalent is

$$P_1 \equiv \text{Alw}\left(\text{replyS1}_{\text{Start}} \wedge \text{Futr}(\text{replyS2}_{\text{Start}}, 1) \Rightarrow \neg \text{SomFi}_{\text{i}}(\text{searchNextS2}_{\text{Start}}, \text{SDSearch})\right).$$

Since in this specific SD no guard is defined for the two operands of $\text{alt}$, both guards are implicitly true, and according to the FFT variant of [Choice], the first operand is always chosen. Consequently, for configurations in line with FFT, the second operand of $\text{alt}$ never gets activated, and the property holds for them. Among the other configurations, it holds for Config1 (since the two replies are separated by a border, they can only occur in the order depicted in the diagram), but not for Config3 and Config5. In the counterexamples provided by Corretto for the two cases in which the property fails, $T_3 = \{\langle\text{replyS1}, \text{replyS2}\rangle, \langle?\text{replyS2}\rangle, \langle?\text{replyS1}\rangle, \langle\text{replyS1}\rangle, \langle\text{searchNextS2}\rangle\}$ is part of the traces, because the sending of $\text{replyS1}$ and $\text{replyS2}$ can be in any order thanks to the choice of weak sequencing. $P_2$ is “If app receives a reply from server1 one time instant before receiving one from server2, app will not send any search request to server2 during SDSearch”; its formal equivalent is

$$P_2 \equiv \text{Alw}\left(\text{replyS1}_{\text{End}} \wedge \text{Futr}(\text{replyS2}_{\text{End}}, 1) \Rightarrow \neg \text{SomFi}_{\text{i}}(\text{searchNextS2}_{\text{Start}}, \text{SDSearch})\right).$$

The property holds for every configuration; that is, for all traces, no matter the interpretation, if there is $?\text{replyS1}$ at time instant $t$ and $?\text{replyS2}$ at $t+1$, then there is no $!\text{searchNextS2}$ at any future time instant in the trace.

Table 6.1: Experimental results with Corretto (times are in seconds).

<table>
<thead>
<tr>
<th></th>
<th>Config1</th>
<th>Config2</th>
<th>Config3</th>
<th>Config4</th>
<th>Config5</th>
<th>Config6</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>UNSAT</td>
<td>UNSAT</td>
<td>SAT</td>
<td>SAT</td>
<td>SAT</td>
<td>SAT</td>
</tr>
<tr>
<td>T2</td>
<td>UNSAT</td>
<td>UNSAT</td>
<td>UNSAT</td>
<td>UNSAT</td>
<td>SAT</td>
<td>SAT</td>
</tr>
<tr>
<td>P1</td>
<td>Verified</td>
<td>Verified</td>
<td>Falsified</td>
<td>Verified</td>
<td>Falsified</td>
<td>Verified</td>
</tr>
<tr>
<td>P2</td>
<td>Verified</td>
<td>Verified</td>
<td>Verified</td>
<td>Verified</td>
<td>Verified</td>
<td>Verified</td>
</tr>
<tr>
<td>TC</td>
<td>UNSAT</td>
<td>UNSAT</td>
<td>UNSAT</td>
<td>UNSAT</td>
<td>SAT</td>
<td>SAT</td>
</tr>
</tbody>
</table>
Corretto also supports time constraints. We exploit this capability to study the real-time-related behavior of SDSearch and the impact of the different configurations. We set the number of loop iterations to 2 in order to ease the understanding of the diagram. The three time constraints state that the transmission of messages updateResultS1 and updateResultS2, that is, the time difference between sending and receiving them, takes at least one time instant and that the whole SDSearch takes no longer than 8 time instants. This is captured by the following annotations added to the SD:

\[
\begin{align*}
(@\text{updateResultS1}_\text{End} - @\text{updateResultS1}_\text{Start}) & \geq 1 \\
(@\text{updateResultS2}_\text{End} - @\text{updateResultS2}_\text{Start}) & \geq 1 \\
(@\text{SDSearch}_\text{End} - @\text{SDsearch}_\text{Start}) & \leq 8
\end{align*}
\]

Table 6.1 shows that these constraints can only be satisfied if Config5 and Config6 are adopted. These are the cases where weak sequencing is used to manage loop iterations.

Our results witness the importance of the semantics adopted for reasoning on SDs. It cannot remain implicit, and the designer must be both aware of the subtle choices and be able to set them.

6.3 Zot

Zot⁴ is an extensible Bounded Model/Satisfiability Checker written in Common Lisp. More precisely, Zot is capable of performing bounded satisfiability checking of formulae written both in LTL (with past operators) and in the propositional, discrete-time fragment of the metric temporal logic TRIO [40], which is equivalent to LTL, but more concise. In fact, TRIO formulae can straightforwardly be translated into LTL formulae, so we use the two temporal logics interchangeably.

The verification process in Zot goes through the following steps: (i) the user writes the specification to be checked as a set of temporal logic formulae (these formulae could also be produced automatically as in Corretto), and selects the plugin and the time bound (i.e., the value of bound $k$) to be used to perform the verification; (ii) depending on the input temporal logic (TRIO or LTL) and the selected plugin, Zot encodes the received specification in a target logic (e.g., Propositional Logic, or Bit-Vector Logic); (iii) Zot feeds the encoded specification to a solver that is capable of handling the target logic; (iv) the result obtained by the solver is parsed back and presented to the user in a textual representation. Zot supports both SAT solvers (e.g., MiniSat [33]) for Propositional Logic, and SMT solvers (e.g., Z3 [61]) for Bit-Vector Logic and decidable fragments of first-order logic.

Zot is a plugin-based tool and every plugin is meant to be the best choice for a specific temporal logic. Figure 6.2 depicts the architecture of the Zot tool. There are interface modules that pass the ultimate formula in appropriate format to the solver and translate back the output produced by the solver by means of the mapping between the APs in the temporal formula and the variables in the unfolded formula.

Zot scripts both in the input model and plugins are a collection of Lisp statements. The following describes the plugins related to this thesis and experimental analysis:

⁴https://github.com/fm-polimi/zot
### Chapter 6. Tools and Experimental Evaluation

#### Figure 6.2: Overall architecture of Zot

<table>
<thead>
<tr>
<th>Zot script (Lisp)</th>
<th>TRIO</th>
<th>LTLB</th>
<th>Logic languages</th>
<th>TRIO</th>
<th>CLTLB</th>
</tr>
</thead>
<tbody>
<tr>
<td>eezot</td>
<td></td>
<td></td>
<td>Zot script</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bezot</td>
<td></td>
<td></td>
<td>plugins</td>
<td></td>
<td></td>
</tr>
<tr>
<td>meezot</td>
<td></td>
<td></td>
<td>solvers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mbeezot</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>zot2cnf</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sat-interface</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>smt-interface</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

####SAT solver (minisat, zchaff, ...)

- **meezot** translates TRIO formulae without arithmetic constraints into the input language of SAT solvers, that is propositional logic. It encodes the metric operator $Dist$, and it is the fastest plugin for the classic encoding of LTL based on eventualities presented in Section 2.5.

- **bvzot** works based on the encoding proposed in Section 5.3 and translates TRIO formulae without arithmetic constraints into $QF_{BV}$ formulae to be fed into an SMT solver.

- **ae2zot** translates CLTL formulae into the input language of SMT solvers that is either a $QF_{LIA}$ (Quantifier-Free Linear Integer Arithmetic) or $QF_{LRA}$ (Quantifier-Free Linear Real Arithmetic) formula. If all the arithmetic constraints are in integer, the output formula (input formula for the SMT solver) is in $QF_{LIA}$, and $QF_{LRA}$ otherwise.

- **ae2bvzot** works based on the encoding proposed in Section 5.5 and translates TRIO formulae into formulae either in a combination of $QF_{BV}$ and $QF_{LIA}$, or $QF_{BV}$ and $QF_{LRA}$ formulae and fed to an SMT solver.

### 6.4 Experimental Evaluation for bvzot

To evaluate the bit-vector-based encoding we compared **bvzot** against three other encodings available in the literature: the classic bounded encoding presented in [16]; the
optimized encoding presented in [56], which has been further improved in [57] and made incremental in [49]; and the encoding optimized for metric temporal logic presented in [71]. The first two encodings are implemented in the well-known NuSMV model checker [23] (in fact, NuSMV implements an optimized, incremental version of the classic encoding of [16]), whereas the third is implemented in the meezot plugin of the Zot tool.

In the rest of this section we will label the experiments carried out with the classic encoding implemented in NuSMV as \textit{bmc}, those performed with the optimized encoding of [57] as \textit{smbc}, those with the incremental version of \textit{sbmc} presented in [49] as \textit{sbmc\_inc}\footnote{While conducting the \textit{sbmc\_inc} experiments we did not activate the completeness checking option since it often slows the verification down, as shown in [49].}, and those performed with the metric encoding implemented in Zot as \textit{meezot} (all these labels come from the commands used in NuSMV and Zot to select the encodings). Note that both NuSMV and Zot support other encodings for LTL/TRIO; we have chosen those mentioned above because further experiments, not reported here, have shown them to be, on average, the most efficient ones for the two tools.

To test the relative efficiency of the four encodings, we applied them to the verification of three case studies, two from the literature and one from previous work of ours. The case studies were chosen mostly for their complexity to highlight the relative strengths and weaknesses of each tool. These three case studies employ a BSC approach, that is, they use temporal logic to describe both the system under verification and the properties to be checked. In all three cases, we performed two kinds of checks. First, we took the temporal logic formula $\phi_S$ describing the system, and we simply checked for its satisfiability. This allowed us to determine whether the specification is realizable or not. As a second type of check, we also provided a logic formula $\phi_P$ capturing the property that the system should have satisfied, and we fed the BSC algorithm with formula $\phi_S \land \neg \phi_P$ to determine whether the property holds for the system or not. We also experimented with different bounds $k$ to analyze how the tools behave when $k$ is increased.

We now briefly introduce the three case studies.

**Kernel Railway Crossing (KRC).** The KRC problem is frequently used for comparing real-time notations and tools [48]. A railway crossing system prevents crossing of the railway by vehicles during passage of a train, by controlling a gate. A temporal logic-based version of the KRC was developed in [71] for benchmarking purposes. It describes one track and one direction of movement of the trains, and it considers an interlocking system. We experimented with two sets of time constants that allow different degrees of nondeterminism, hereafter denoted as \textit{krc1} and \textit{krc2}. The level of nondeterminism is increased by using bigger time constants, e.g., the time for a train to go through the railway crossing, which increase the number of possible combinations of events in the system. We also carried out formal verification with two properties of interest: a safety property that says that as long as a train is in the critical region the gate is closed ($P1$); and a utility property that states that the gate must be open when it is safe to do so (i.e., the gate should not be closed when unnecessary), where the notion of “safe” is captured through suitable time constants ($P2$).

**Fischer’s Protocol.** This is a classic algorithm for granting exclusive access to a resource that is shared among many processes. Fischer’s protocol is a typical benchmark
for verification tools capable of dealing with real-time constraints. The version we used for our tests, where the specification of the system is described through temporal logic formulae, is taken from [71]; it includes 4 processes, and the delay that a process waits after sending a request, which is the key parameter in Fischer’s protocol, is 5 time instants. We performed formal verification of a safety property that states that it is never the case that two processes are simultaneously in their critical sections (P1).

**Verification of UML Diagrams.** In our tests we used Corretto to produce the formalism for the example diagrams shown in Figure 6.1, which describe the behavior of an application that pings two servers, and then sends queries to the server that responds first. The model comprises a loop, and we performed tests on two versions of the system, called sdserver12 and sdserver13, where the number of iterations in the loops is 2 and 3, respectively. We also performed formal verification on the example system using property P1 defined in Section 6.2. The formal semantics are produced according Config5.

To compare the performances of the bmc, sbmc, sbmc_inc, meezot and bvzot encodings, we built a simple translation tool that converts specifications written in the Zot input language such as those used in [71] and [7] into the input language of NuSMV.

![Figure 6.3: Time/Memory Comparison for KRC1 (SAT and P1).](image)

Figures 6.3-6.9 show the time (in seconds) and memory (in MBs) consumed in each of the experiments we performed. Note that if no bar is visible, and no error tag is reported, this means that the number is very small.6

For example, Figure 6.3 shows the time/memory consumption for each encoding (bmc, sbmc, sbmc_inc, meezot, and bvzot) for the various checks on example krc1: simple satisfiability checking with maximum bound \( k = 30 \) (sat_30) and verification of property P1 with maximum bound \( k = 30 \) (P1_30), \( k = 60 \) (P1_60) and \( k = 90 \).

---

6.4. Experimental Evaluation for bvzot

![Graph](image)

*Figure 6.4: Time/Memory Comparison for KRC1 (P2).*

![Graph](image)

*Figure 6.5: Time/Memory Comparison for KRC2 (SAT and P1).*

(P1_90), respectively. The role of the “maximum bound” is the following: for a given maximum bound \(k\), the tools iteratively (possibly incrementally) try to find an ultimately periodic model \(\alpha \beta^\omega\) where the length of \(\alpha \beta\) is 1, 2, \ldots, \(k\). As soon as a model is found, the search stops, and the model is output; if no model is found for any bound up to \(k\), the search stops at \(k\) and the formula is declared unsatisfiable.

All the runs reported in Figures 6.3-6.9 had a time limit of 1 hour and a memory limit of 4GB RAM; that is, if the verification took longer than 1 hour or occupied
more than 4GB of RAM, it was stopped. Hence, the possible outcomes of a run are **satisfiable**, **unsatisfiable**, **out of time** (TO), and **out of memory** (MO). In addition, in some cases the tool stopped with a **segmentation fault** (SF) error, and in others with **heap exhausted** (HE) while pre-processing the specification to produce the encoding.

All the experiments were carried out on a Linux desktop machine with a 3.4 GHz Intel® Core™ i7-4770 CPU and 8 GB RAM\(^7\). The NuSMV version was 2.5.4. The SAT

\(^7\)bzvat, along with the code for all the experiments, is available on the Zot repository [82]
6.4. Experimental Evaluation for bvzot

![Time/Memory Comparison for SDServer12 (SAT and P1).](image1)

![Time/Memory Comparison for SDServer13 (SAT and P1).](image2)

and SMT solvers used with Zot were, respectively, MiniSat version 2.2 and Z3 version 4.3.2.

As the figures show, among the algorithms implemented in NuSMV, \texttt{sbmc\_inc} is the most memory efficient, while \texttt{sbmc} is the fastest. There are six models that \texttt{sbmc\_inc} can afford to verify, while \texttt{sbmc} fails. However, for the models both encodings can afford, \texttt{sbmc} is faster than \texttt{sbmc\_inc}.

When performing checks that require small bounds, such as the satisfiability checks,
Chapter 6. Tools and Experimental Evaluation

*bvzot* is only occasionally more efficient than the other tools. However, as the models and bounds grow in size, *bvzot* demonstrates its strengths. For example, when proving properties for the KRC version with the highest level of nondeterminism, i.e., *krc2*, *bvzot* is the only tool able to explore all the bounds up to 90, and it is faster than the others when the time bound is kept smaller (30 or 60). Similar results hold for the verification of properties on the UML diagrams, whose formalization in temporal logic is in fact the biggest specification that we have tested due to the necessity of capturing all the possible sequences of events in the Sequence Diagram.

However, we must highlight that in the case of Fischer’s protocol, *sbmc* is the most efficient encoding time-wise, whereas *bvzot* is often the one with the least memory consumption.

![Diagram](image)

**Figure 6.10:** Comparison between classic and bit-vector-based encoding of LTL formulae.

All in all, we can conclude that the experimental results show a promising ability by *bvzot* to scale up as the size of the specification and of the time bound increase. Further gains could also be obtained by adapting some of the optimizations presented in [57] in *bvzot*.

The efficiency of our encoding mainly owes to the word-level simplification of the Bit-Vector Logic formulae that capture the temporal operators of the original CLTL formula, shown in Figure 6.10. In fact, in the classic Boolean-based encodings there are groups of Boolean variables capturing the value of atomic propositions, much like our bit-vectors, but the solver is blind to their interrelations, because constraints are asserted at the bit-level. Therefore, in Boolean-based encodings there are no simplifications attempted at the level of the whole word, whereas modern SMT solvers efficiently handle such simplifications when atomic propositions are introduced as bit-vectors. For

---

8For each satisfiability experiment, the length of the smallest model found was the same for each tool: 1 for the *krc* examples (the execution in which nothing happens, that is, no train enters the crossing, is admissible), 29 for Fischer’s protocol, 10 for *sds serverl2*, and 13 for *sds serverl3*. 

74
example, in the case of the S and U operators, we use binary additions and bitwise operations to provide a very concise encoding of the temporal operators.

6.5 Experimental Evaluation for ae2bvzot

Plugin ae2zot is capable of deciding the satisfiability of both CLTL and CLTLoc formulae. To achieve this, the plugin implements the bounded approach described in [10, 12, 14]. To solve the satisfiability problem for a formula $\phi$ that belongs to either CLTL or CLTLoc, the ae2zot plugin unfolds $\phi$ over a finite model of length $k + 2$, where the last state of the model is the repetition of a previous one, and translates the unfolded formula into a formula of a suitable decidable logic. The target logic depends on the nature of $\phi$. If $\phi$ is a CLTL formula where the domain of the variables is $\mathbb{N}$ (i.e., $D = (\mathbb{N}, \{<,=\})$) or $\mathbb{Z}$, then the target logic is QF_LIA; if $\phi$ is either a CLTL formula where the values of variables are in $\mathbb{R}$, or it is a CLTLoc formula, then the target logic is QF_LRA (i.e., the arithmetic part is over the reals). The resulting QF_LIA/QF_LRA formula is fed to an off-the-shelf SMT solver such as z3 [64].

To improve with respect to the ae2zot plugin, we have separated the encoding of the temporal operators, which is now done through the bit-vector-based approach presented in Section 5.3, from the representation of the arithmetic variables. We have called the resulting plugin, which mixes QF_BV with QF_LIA/QF_LRA, ae2bvzot.

To maximize the efficiency of an SMT solver, one needs to configure it properly by indicating what tactics should be used to solve the given problem. Some preliminary experiments we carried out, and discussed below, showed that this is especially true when the problem to be solved is expressed through the combination of different logics. There are numerous configuration parameters in SMT solvers, which in many cases are documented rather briefly, and trying all of them is almost infeasible. Choosing the most efficient configuration out of the many possible ones was a trial-and-error process guided by our intuition of what reasonable tactics could be. We do not claim that this process led us to the absolute best possible configuration of the SMT solver for checking CLTL/CLTLoc specifications. It is possible that even better configurations can be found by further studying the shape of the SMT problems that are produced by the bounded decision procedure for CLTL/CLTLoc, but we leave this for future work.

In the set-up phase of our experiments, then, we tried many combinations of different tactics to be used by the ae2zot and ae2bvzot plugins when invoking the SMT solver (z3 in our case), to find the best ones in the two cases. The result was that we could hardly improve on the default configuration of z3 when a single logic was involved (i.e., in the ae2zot case), but that the efficiency of the verification could be increased significantly with respect to the default configuration when multiple logics were used (i.e., for the ae2bvzot plugin).

Finally, we configured the ae2bvzot plugin so that, when z3 is invoked, the tactics are applied in the following order: first we perform simplification and elimination of variables through the solving of equations; then, the solver performs bit-blasting to reduce the bit-vector expression into a Boolean satisfiability problem; and finally the solver uses a SAT-based tactic on this problem.

In the following, we first briefly present the CLTL and CLTLoc case studies over which we compared the performance of the ae2zot and ae2bvzot plugins; then, we
show the experimental results, and finally we draw some considerations on the results of the comparison.

We performed our comparison over four groups of examples, two concerning CLTL, and two concerning CLTLoc. For both CLTL and CLTLoc the corresponding two groups of examples differ in the way the CLTL/CLTLoc models have been produced: in one group, the models have been produced by hand from an informal description; in the second group, the temporal logic model has been automatically generated from another, formal or semi-formal, description.

More precisely, in the case of CLTL, we built by hand the models for two well-known examples, a bubblesort-style sorting algorithm, and the leader election protocol introduced in Section 2.6. We have also used specifications automatically generated from multi-diagram UML models using the approach described in [6].

In the case of CLTLoc, the model built by hand is a standard timed lamp which has been used many times for testing the performance of verification tools (see, e.g., [71]), and which has been given a CLTLoc description in [12]. The bulk of the CLTLoc experiments, however, used models that have been created using the transformation from continuous-time MITL specifications that has been defined in [15]. In effect, these are experiments in verification of continuous-time MITL models, which exploit CLTLoc as intermediate language and use the corresponding decision procedure.

We remark that in all experiments the approach is entirely logic-based, and the verification is always an instance of the bounded satisfiability checking problem. That is, in all our examples both the system being analyzed and the property to be checked (if any) are expressed in temporal logic. This differs from so-called bounded model checking mechanisms, where the system is expressed in some kind of operational formalism, typically labeled transition systems.

In general, we perform two kinds of experiments: consistency checking ones (SAT), where we feed the verification tool with only the system model, without any property to be verified, and ask for an execution trace that witnesses the feasibility of the model (i.e., we check that the system has at least one admissible execution, hence it is not inconsistent); and classic property verification experiments, where we feed the tool with the system and the property to be verified (both described through temporal logic formulae), and we check whether the latter holds for the former or not (in which case the tool returns a trace witnessing the violation).

Let us briefly introduce the case studies we used in our experiments.

**Sorting.** This model specifies a sorting process of an array of fixed size \(N\), using CLTL over \(\mathcal{D} = (\mathbb{Z}, \{<, =\})\). This model is introduced in [10]. We indicate by \(b, a \in \mathbb{Z}^N\) the array we want to sort and the array during each step of sorting, respectively, and by \(b_i\) the \(i\)-th element in \(b\) (similarly for \(a_i\)). The model consists in a sorting process that nondeterministically chooses an index \(1 \leq s \leq N - 1\) such that \(a_s > a_{s+1}\) and swaps \(a_s\) with \(a_{s+1}\). The sorting process keeps swapping unsorted adjacent elements until the whole array is sorted (Formula (2.1) is an example of CLTL formula capturing the swapping mechanism). The following is a sample property to be checked that says that eventually the array gets sorted:

\[
F \left( \bigwedge_{i=1}^{N-1} (a_i \leq a_{i+1}) \land \bigwedge_{i=1}^{N} \bigvee_{j=1}^{N} a_i = b_i \right). \tag{6.1}
\]
6.5. Experimental Evaluation for ae2bvzot

In addition to the model in which the elements to be sorted are arbitrary integer numbers, we also performed experiments on a model which is built upon the same CLTL formulae, but where elements are real-valued; that is, in this second case we have that that $b, a \in \mathbb{R}^N$, and $D = (\mathbb{R}, \{<, =\})$.

We also use a generalized version of this sorting process, in which instead of swapping only adjacent values $a_s, a_{s+1}$, the algorithm swaps $a_s$ with possibly any $a_z$, provided that $z > s$ and $a_s > a_z$. In other words, any pair of unsorted elements can be nondeterministically selected for swapping.

**Leader Election Protocol.** This case study consists in the CLTL model (with $D = (\mathbb{Z}, \{<, =\})$) of the leader election protocol described in Section 2.6. It is a CLTL version of the Promela model included in the Spin distribution [77].

**Car Collision Avoidance System (CCAS).** This example is taken from [6]. It is originally described in UML, then translated into CLTL through Corretto. The example concerns a system that detects the distance of the vehicle on which it is installed, with respect to other objects such as cars and pedestrians. The distance between the car and the external objects is read by a sensor, which sends the data to the CCAS main module every 100 ms through the system bus. When the distance between the car and the external objects is greater than or equal to 2 meters the CCAS should perform no action. When the distance becomes strictly less than 2 meters the CCAS switches to the warning state. If the distance remains in the warning state for more than 300 ms and the distance is still less than 2 meters, the CCAS must brake the car. In this case, we use CLTL with $D = (\mathbb{Z}, \{<, =\})$ to capture the data that is sent by the sensor to the main module, and that triggers the action. On this system, we want to prove the property that “if the distance remained less than 2 meters for $T$ time units, then the system braked within those same $T$ time units”, where $T$ is a fixed positive integer, and each (discrete) time unit corresponds to 10 ms.

**Leader Election Protocol - UML version.** This is again the leader election protocol, but first modeled in UML, then translated into CLTL.

**Timed Lamp - CLTLoc version.** This example is taken from [12, 14]. It consists of a lamp that is controlled by two buttons, $ON$ and $OFF$, which cannot be pressed simultaneously. The lamp can be either on or off. When $ON$ (resp. $OFF$) is pressed, the lamp is immediately turned on (resp. off). After $ON$ is pressed, if no more buttons are pressed, it will automatically turn off with a delay $\Delta$, a positive real constant. If the $ON$ button is pressed again before the timeout expires, then the timeout is extended by a new delay $\Delta$. Formula (2.2) is an example of CLTLoc formula for the timed lamp, where $\Delta = 5$. In this case, we check properties such as “the light never stays on for longer than $\Delta$ time units” and “if at some point the light stays on for longer than $\Delta$ time units, then $ON$ is eventually pressed, and it is pressed again before $\Delta$ time units”.

**Timed Lamp - Continuous time MITL specification.** This example is also taken from [12, 14]. It is a pure MITL specification of the previously described behavior of the timed lamp over so-called continuous time signals. In this example, we exploit the MITL-to-CLTLoc satisfiability-preserving translation described in [15] to carry out the verification.

**Continuous time MITL specifications.** These are examples from [15], and exploit the aforementioned MITL-to-CLTLoc satisfiability-preserving translation. They concern
“events” occurring in single instants over the real line (for example, predicate $p$ occurring exactly when the current instant is a multiple of 100). We impose constraints such as “$q$ must occur within 1 time unit (in the future or in the past) of $p$,” then check properties such as “after each $q$ there is another $q$ within 100 time units”. The examples include also the so-called “counting” operators, which allow users to state properties such as “$q$ will hold at least $n$ times in the next interval of length 1” (with $n$ a constant, for example 2). Tables 6.2-6.8 show the result (R) of the verification, which can be satisfiable (S) or unsatisfiable (U), the time (T) in seconds and memory (M) in MBs consumed in each of the experiments we performed$^9$. More precisely, Tables 6.2-6.5 show the results for the experiments carried out with CLTL, whereas Tables 6.6-6.8 present those where the logic used was CLTLoc.

To aid the reader in getting a quick overview of the results, we colored the cells related to the ae2bvzot tool according to the following scheme. If $tr$ (resp., $mr$) is the ratio between the time taken (resp., memory used) by ae2zot and that taken by ae2bvzot, then the color of the corresponding cell is the following:

- if $tr \geq 2$ or $mr \geq 1.5$ (i.e., ae2bvzot is at least twice as fast as ae2zot, or occupies less than 2/3 the memory), we color the cell dark green;
- if $1.1 \leq tr < 2$ or $1.1 \leq mr < 1.5$, light green;
- if $0.91 < tr < 1.1$ or $0.91 < mr < 1.1$, no color;
- if $0.5 < tr \leq 0.91$ or $0.66 < mr < 0.91$, light red;
- if $tr < 0.5$ or $mr \leq 0.66$, dark red.

Let us briefly explain the meaning of the identifiers used in the tables.$^{10}$

In Table 6.2, *Sorting1-* rows capture the experiments with the model of the sorting algorithm where elements are integers ($b,a \in \mathbb{Z}^N$), whereas in *Sorting1-R-* rows elements are real-valued ($b,a \in \mathbb{R}^N$). The *Sorting2-N-* identifier, instead, stands for the generalized version of the algorithm that can swap arbitrary numbers. In all cases the postfix (*) is a number that identifies the check that was performed (e.g., pure SAT checking to see if the model is feasible, or checking of a specific property), whereas $N$ (with $N \in \{5, 6, 7, 8\}$) is the length of the array. Similarly, *Leader-N-* and *UML-Leader-N-* identify the experiments performed using the model of leader election protocol described, respectively, “natively” in CLTL and through UML diagrams first; $N$ corresponds to the number of elements in the ring, and the postfix identifies the check performed. Finally, rows labeled *CCAS-N-* contain the results (pure satisfiability, verification of property $p_1$ and $p_2$) for the experiments with the CCAS example. There are 5 versions of this model, identified by number $N$, that differ from one another in the values of some temporal bounds, such as the maximum duration that the system stays in the warning state, the delay with which the brakes are activated, and the time constants $T$ in the property checked.

In the case of the CLTLoc experiments of Tables 6.6-6.8, rows labeled *Lamp-CLTLoc-* correspond to the experiments carried out using the native CLTLoc specification of the

---

$^9$The code for all the experiments is available at http://home.deib.polimi.it/pourhashem.kallehbasti/sac-2016.php

$^{10}$An extended, informal, description of each model – and of the verification performed – referenced in the tables can be found at http://home.deib.polimi.it/pourhashem.kallehbasti/ModelDescriptions.pdf
Table 6.2: Comparison between ae2zot and ae2bvzot on the Sorting models.

<table>
<thead>
<tr>
<th>Model</th>
<th>Tool</th>
<th>R</th>
<th>ae2zot</th>
<th>ae2bvzot</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>T(s)</td>
<td>M(MB)</td>
</tr>
<tr>
<td>Sorting1-1</td>
<td>S</td>
<td>2</td>
<td>155</td>
<td>1</td>
</tr>
<tr>
<td>Sorting1-2</td>
<td>U</td>
<td>99</td>
<td>181</td>
<td>48</td>
</tr>
<tr>
<td>Sorting1-3</td>
<td>S</td>
<td>10</td>
<td>175</td>
<td>3</td>
</tr>
<tr>
<td>Sorting1-R-1</td>
<td>S</td>
<td>6</td>
<td>180</td>
<td>1</td>
</tr>
<tr>
<td>Sorting1-R-2</td>
<td>U</td>
<td>90</td>
<td>209</td>
<td>6</td>
</tr>
<tr>
<td>Sorting1-R-3</td>
<td>S</td>
<td>13</td>
<td>209</td>
<td>2</td>
</tr>
<tr>
<td>Sorting2-5-1</td>
<td>S</td>
<td>13</td>
<td>196</td>
<td>5</td>
</tr>
<tr>
<td>Sorting2-5-2</td>
<td>U</td>
<td>34</td>
<td>184</td>
<td>4</td>
</tr>
<tr>
<td>Sorting2-5-3</td>
<td>S</td>
<td>67</td>
<td>220</td>
<td>19</td>
</tr>
<tr>
<td>Sorting2-5-4</td>
<td>S</td>
<td>38</td>
<td>208</td>
<td>21</td>
</tr>
<tr>
<td>Sorting2-5-5</td>
<td>S</td>
<td>11</td>
<td>193</td>
<td>15</td>
</tr>
<tr>
<td>Sorting2-5-6</td>
<td>U</td>
<td>TO</td>
<td>TO</td>
<td>1233</td>
</tr>
<tr>
<td>Sorting2-6-1</td>
<td>S</td>
<td>46</td>
<td>227</td>
<td>20</td>
</tr>
<tr>
<td>Sorting2-6-2</td>
<td>U</td>
<td>1219</td>
<td>227</td>
<td>342</td>
</tr>
<tr>
<td>Sorting2-6-3</td>
<td>S</td>
<td>160</td>
<td>262</td>
<td>51</td>
</tr>
<tr>
<td>Sorting2-6-4</td>
<td>S</td>
<td>204</td>
<td>250</td>
<td>44</td>
</tr>
<tr>
<td>Sorting2-6-5</td>
<td>S</td>
<td>47</td>
<td>223</td>
<td>23</td>
</tr>
<tr>
<td>Sorting2-6-6</td>
<td>-</td>
<td>TO</td>
<td>TO</td>
<td>TO</td>
</tr>
<tr>
<td>Sorting2-7-1</td>
<td>S</td>
<td>51</td>
<td>264</td>
<td>42</td>
</tr>
<tr>
<td>Sorting2-7-2</td>
<td>-</td>
<td>TO</td>
<td>TO</td>
<td>TO</td>
</tr>
<tr>
<td>Sorting2-7-3</td>
<td>S</td>
<td>448</td>
<td>334</td>
<td>61</td>
</tr>
<tr>
<td>Sorting2-7-4</td>
<td>S</td>
<td>253</td>
<td>301</td>
<td>61</td>
</tr>
<tr>
<td>Sorting2-7-5</td>
<td>S</td>
<td>270</td>
<td>272</td>
<td>49</td>
</tr>
<tr>
<td>Sorting2-7-6</td>
<td>-</td>
<td>TO</td>
<td>TO</td>
<td>TO</td>
</tr>
<tr>
<td>Sorting2-8-1</td>
<td>S</td>
<td>1041</td>
<td>399</td>
<td>104</td>
</tr>
<tr>
<td>Sorting2-8-2</td>
<td>-</td>
<td>TO</td>
<td>TO</td>
<td>TO</td>
</tr>
<tr>
<td>Sorting2-8-3</td>
<td>S</td>
<td>1338</td>
<td>467</td>
<td>257</td>
</tr>
<tr>
<td>Sorting2-8-4</td>
<td>S</td>
<td>247</td>
<td>425</td>
<td>171</td>
</tr>
<tr>
<td>Sorting2-8-5</td>
<td>S</td>
<td>1203</td>
<td>372</td>
<td>131</td>
</tr>
<tr>
<td>Sorting2-8-6</td>
<td>-</td>
<td>TO</td>
<td>TO</td>
<td>TO</td>
</tr>
</tbody>
</table>
timed lamp, while Lamp-MITL-* are those where the model of the timed lamp is originally described through continuous-time MITL formulae translated into equisatisfiable CLTLoc formulae.

Rows labeled with Spikes-* and Wave-* are verification experiments of MITL models capturing particular behaviors where phenomena behave as events ("spikes") or as rectangular waves. Finally, labels CountingX-*, with X ∈ {1, 2, 3}, identify experiments starting from MITL specifications that also include the “counting” operator.

All the experiments were carried out on a Linux desktop machine with a 3.4 GHz Intel® Core™ i7-4770 CPU and 8 GB RAM. All the reported runs had a timeout of 1 hour; i.e., if the verification took longer than 1 hour, it was aborted (TO). The models for the leader election protocol, however, are more time consuming as the number of nodes in the ring increases. Hence, in this case, to make the comparison more meaningful, the time limit was set to 10 hours.

From the experimental results, we can draw some considerations on the effectiveness of the new verification tool, and of the kinds of problems for which ae2bvzot seems particularly well suited.

First of all, we remark that the main feature of the ae2bvzot plugin is that it combines two different logics, Bit-Vector Logic for capturing the behavior of the temporal operators and arithmetic constraints for the first-order variables. Since SMT solvers do not support a logic that combines both the theory of bit-vectors and that of integer/real numbers, they do not have tactics that are specific for the combination of the two logics, so the solver needs to be guided in what tactics to apply to the problem to obtain the best results. This, in turn, suggests that in some cases the interplay between logics makes the solving less efficient than when using one single logic to capture all aspects, both arithmetic and temporal (as it is the case in ae2zot, which uses either only QF_LIA or only QF_LRA as target logic). For example, the position of the loop back is frequently used in the encoding and acts like a bottleneck in combining different layers/logics, since it appears as an integer variable in the arithmetic layer, and as a bit-vector in the temporal one. This emerges also from our experiments, where it seems that, when the arithmetic part of the model becomes more and more significant, the gains obtained with the ae2bvzot plugin decrease or disappear entirely.

For example, the comparison between ae2zot and ae2bvzot becomes in general less favorable for the latter in the CLTLoc examples that have been derived by translation from continuous-time MITL specifications, as evidenced in Tables 6.6-6.8 with respect to Tables 6.2-6.5. In these cases, in fact, the number of arithmetic variables becomes considerable (multiple clocks, i.e., arithmetic variables, are introduced for each subformula of the original MITL formula); in addition, to manage the advancement of time the solver needs to take into account not only comparisons between values, but also more complicated operations such as addition of delays.

When the temporal, propositional part is predominant, instead, as in most of the CLTL case studies shown in Tables 6.2-6.5, the gains that have been obtained by the purely bit-vector-based encoding manifest themselves also in the ae2bvzot plugin. In these models, especially the sorting and leader election cases, the arithmetic part is simpler, as it is essentially confined to establishing comparisons between values and to perform value assignments.

The CCAS case study requires a separate discussion. In fact, on this example the
6.5. Experimental Evaluation for \texttt{ae2bvzot}

\texttt{ae2zot} plugin mostly outperforms \texttt{ae2bvzot}. We remark however that, unlike the sorting and the leader election examples, where the nature of the models is such that the various instances differ in their structures (the size of the array and the size of the ring of processes change, hence the number of arithmetic variables also changes), the various versions of the CCAS all have the same components and variables, and they differ only in the values of the temporal constants involved in the model. Hence, it is natural that, if \texttt{ae2zot} is more efficient in one case (say, \texttt{CCAS-1-\*}), so is for the other cases. As for the reason why \texttt{ae2zot} is the better plugin for this case, we conjecture that it depends on the fact that the behavior of the arithmetic variables in the CCAS is rather rigid, as they are constrained to be piecewise constant, which in turn increases the interplay between the arithmetic and temporal parts of the model.

Finally, we remark that the \texttt{ae2bvzot} is, in many CLTL test, slightly more memory consuming than \texttt{ae2zot}. However, the difference is, especially in the sorting case, still rather limited (mostly around 10\% worsening), with a steep increase in efficiency.
### Table 6.3: Comparison between *ae2zot* and *ae2bvzot* on the CCAS models.

<table>
<thead>
<tr>
<th>Model</th>
<th>Tool</th>
<th>R</th>
<th>ae2zot T(s)</th>
<th>ae2zot M(MB)</th>
<th>ae2bvzot T(s)</th>
<th>ae2bvzot M(MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CCAS-1-sat</td>
<td>S</td>
<td>15</td>
<td>461</td>
<td></td>
<td>17</td>
<td>574</td>
</tr>
<tr>
<td>CCAS-1-p1</td>
<td>S</td>
<td>30</td>
<td>565</td>
<td></td>
<td>23</td>
<td>635</td>
</tr>
<tr>
<td>CCAS-1-p2</td>
<td>U</td>
<td>42</td>
<td>485</td>
<td></td>
<td>60</td>
<td>638</td>
</tr>
<tr>
<td>CCAS-2-sat</td>
<td>S</td>
<td>14</td>
<td>463</td>
<td></td>
<td>16</td>
<td>591</td>
</tr>
<tr>
<td>CCAS-2-p1</td>
<td>S</td>
<td>39</td>
<td>538</td>
<td></td>
<td>19</td>
<td>639</td>
</tr>
<tr>
<td>CCAS-2-p2</td>
<td>U</td>
<td>42</td>
<td>499</td>
<td></td>
<td>57</td>
<td>644</td>
</tr>
<tr>
<td>CCAS-3-sat</td>
<td>S</td>
<td>16</td>
<td>466</td>
<td></td>
<td>17</td>
<td>593</td>
</tr>
<tr>
<td>CCAS-3-p1</td>
<td>S</td>
<td>32</td>
<td>548</td>
<td></td>
<td>19</td>
<td>649</td>
</tr>
<tr>
<td>CCAS-3-p2</td>
<td>U</td>
<td>41</td>
<td>511</td>
<td></td>
<td>55</td>
<td>660</td>
</tr>
<tr>
<td>CCAS-4-sat</td>
<td>S</td>
<td>14</td>
<td>460</td>
<td></td>
<td>30</td>
<td>572</td>
</tr>
<tr>
<td>CCAS-4-p1</td>
<td>S</td>
<td>39</td>
<td>535</td>
<td></td>
<td>20</td>
<td>638</td>
</tr>
<tr>
<td>CCAS-4-p2</td>
<td>U</td>
<td>41</td>
<td>492</td>
<td></td>
<td>36</td>
<td>638</td>
</tr>
<tr>
<td>CCAS-5-sat</td>
<td>S</td>
<td>16</td>
<td>463</td>
<td></td>
<td>32</td>
<td>590</td>
</tr>
<tr>
<td>CCAS-5-p1</td>
<td>S</td>
<td>29</td>
<td>539</td>
<td></td>
<td>18</td>
<td>641</td>
</tr>
<tr>
<td>CCAS-5-p2</td>
<td>U</td>
<td>34</td>
<td>503</td>
<td></td>
<td>41</td>
<td>650</td>
</tr>
</tbody>
</table>
### 6.5. Experimental Evaluation for ae2bvzot

Table 6.4: Comparison between ae2zot and ae2bvzot on the Leader models.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Model</th>
<th>R</th>
<th>ae2zot</th>
<th>ae2bvzot</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>T(s)</td>
<td>M(MB)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>T(s)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>M(MB)</td>
</tr>
</tbody>
</table>

| Leader-5-sat | S | 28 | 381 | 2 | 254 |
| Leader-5-p1  | U | 59 | 309 | 2 | 237 |
| Leader-5-p2  | U | 19 | 333 | 5 | 257 |
| Leader-10-sat| S | 4997 | 2882 | 7 | 412 |
| Leader-10-p1 | U | 1016 | 1084 | 67 | 612 |
| Leader-10-p2 | U | 19837 | 1650 | 161 | 667 |
| Leader-12-sat| S | 15776 | 5535 | 51 | 792 |
| Leader-12-p1 | U | 3686 | 1660 | 400 | 1025 |
| Leader-12-p2 | U | TO | TO | 938 | 1131 |
| Leader-14-sat| S | TO | TO | 45 | 789 |
| Leader-14-p1 | U | 18830 | 3128 | 2183 | 1910 |
| Leader-14-p2 | U | TO | TO | 7395 | 2930 |
| Leader-15-sat| S | TO | TO | 85 | 1282 |
| Leader-15-p1 | U | TO | TO | 6797 | 3636 |
| Leader-15-p2 | U | TO | TO | 18526 | 4679 |
| Leader-16-sat| S | TO | TO | 135 | 1450 |
| Leader-16-p1 | U | TO | TO | 14558 | 4143 |
| Leader-16-p2 | - | TO | TO | TO | TO |
Table 6.5: Comparison between ae2zot and ae2bvzot on the UML-Leader models.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Model</th>
<th>R</th>
<th>T(s)</th>
<th>M(MB)</th>
<th>T(s)</th>
<th>M(MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>UML-Leader-5-sat</td>
<td>S</td>
<td>96</td>
<td>728</td>
<td>10</td>
<td>434</td>
</tr>
<tr>
<td></td>
<td>UML-Leader-5-p1</td>
<td>U</td>
<td>272</td>
<td>822</td>
<td>11</td>
<td>446</td>
</tr>
<tr>
<td></td>
<td>UML-Leader-5-p2</td>
<td>U</td>
<td>223</td>
<td>798</td>
<td>11</td>
<td>428</td>
</tr>
<tr>
<td></td>
<td>UML-Leader-7-sat</td>
<td>S</td>
<td>559</td>
<td>1233</td>
<td>51</td>
<td>803</td>
</tr>
<tr>
<td></td>
<td>UML-Leader-7-p1</td>
<td>U</td>
<td>897</td>
<td>1381</td>
<td>76</td>
<td>713</td>
</tr>
<tr>
<td></td>
<td>UML-Leader-7-p2</td>
<td>U</td>
<td>929</td>
<td>1417</td>
<td>38</td>
<td>636</td>
</tr>
<tr>
<td></td>
<td>UML-Leader-9-sat</td>
<td>S</td>
<td>2175</td>
<td>2384</td>
<td>395</td>
<td>1304</td>
</tr>
<tr>
<td></td>
<td>UML-Leader-9-p1</td>
<td>U</td>
<td>4390</td>
<td>2736</td>
<td>162</td>
<td>954</td>
</tr>
<tr>
<td></td>
<td>UML-Leader-9-p2</td>
<td>U</td>
<td>4933</td>
<td>2900</td>
<td>76</td>
<td>797</td>
</tr>
<tr>
<td></td>
<td>UML-Leader-10-sat</td>
<td>S</td>
<td>5370</td>
<td>2974</td>
<td>117</td>
<td>1228</td>
</tr>
<tr>
<td></td>
<td>UML-Leader-10-p1</td>
<td>U</td>
<td>9918</td>
<td>3822</td>
<td>452</td>
<td>1298</td>
</tr>
<tr>
<td></td>
<td>UML-Leader-10-p2</td>
<td>U</td>
<td>12656</td>
<td>3871</td>
<td>232</td>
<td>987</td>
</tr>
<tr>
<td></td>
<td>UML-Leader-12-sat</td>
<td>S</td>
<td>TO</td>
<td>TO</td>
<td>1465</td>
<td>3398</td>
</tr>
<tr>
<td></td>
<td>UML-Leader-12-p1</td>
<td>U</td>
<td>TO</td>
<td>TO</td>
<td>4049</td>
<td>2852</td>
</tr>
<tr>
<td></td>
<td>UML-Leader-12-p2</td>
<td>U</td>
<td>TO</td>
<td>TO</td>
<td>4264</td>
<td>3215</td>
</tr>
<tr>
<td></td>
<td>UML-Leader-13-sat</td>
<td>S</td>
<td>TO</td>
<td>TO</td>
<td>124</td>
<td>1949</td>
</tr>
<tr>
<td></td>
<td>UML-Leader-13-p1</td>
<td>U</td>
<td>TO</td>
<td>TO</td>
<td>7012</td>
<td>3643</td>
</tr>
<tr>
<td></td>
<td>UML-Leader-13-p2</td>
<td>U</td>
<td>TO</td>
<td>TO</td>
<td>8467</td>
<td>4008</td>
</tr>
<tr>
<td></td>
<td>UML-Leader-14-sat</td>
<td>S</td>
<td>TO</td>
<td>TO</td>
<td>2651</td>
<td>4936</td>
</tr>
<tr>
<td></td>
<td>UML-Leader-14-p1</td>
<td>U</td>
<td>TO</td>
<td>TO</td>
<td>12066</td>
<td>4305</td>
</tr>
<tr>
<td></td>
<td>UML-Leader-14-p2</td>
<td>U</td>
<td>TO</td>
<td>TO</td>
<td>25674</td>
<td>5952</td>
</tr>
</tbody>
</table>

Table 6.6: Comparison between ae2zot and ae2bvzot on the Lamp models.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Model</th>
<th>R</th>
<th>T(s)</th>
<th>M(MB)</th>
<th>T(s)</th>
<th>M(MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Lamp-CLTLoc-1</td>
<td>S</td>
<td>1</td>
<td>156</td>
<td>1</td>
<td>148</td>
</tr>
<tr>
<td></td>
<td>Lamp-CLTLoc-2</td>
<td>U</td>
<td>1</td>
<td>159</td>
<td>2</td>
<td>151</td>
</tr>
<tr>
<td></td>
<td>Lamp-CLTLoc-3</td>
<td>S</td>
<td>2</td>
<td>157</td>
<td>0</td>
<td>149</td>
</tr>
<tr>
<td></td>
<td>Lamp-CLTLoc-4</td>
<td>U</td>
<td>19</td>
<td>167</td>
<td>25</td>
<td>157</td>
</tr>
<tr>
<td></td>
<td>Lamp-CLTLoc-5</td>
<td>U</td>
<td>18</td>
<td>160</td>
<td>17</td>
<td>154</td>
</tr>
<tr>
<td></td>
<td>Lamp-MITL-1</td>
<td>S</td>
<td>1</td>
<td>187</td>
<td>1</td>
<td>145</td>
</tr>
<tr>
<td></td>
<td>Lamp-MITL-2</td>
<td>S</td>
<td>1</td>
<td>185</td>
<td>0</td>
<td>156</td>
</tr>
<tr>
<td></td>
<td>Lamp-MITL-3</td>
<td>S</td>
<td>3</td>
<td>245</td>
<td>2</td>
<td>187</td>
</tr>
<tr>
<td></td>
<td>Lamp-MITL-4</td>
<td>U</td>
<td>98</td>
<td>282</td>
<td>55</td>
<td>223</td>
</tr>
<tr>
<td></td>
<td>Lamp-MITL-5</td>
<td>S</td>
<td>49</td>
<td>669</td>
<td>26</td>
<td>663</td>
</tr>
<tr>
<td></td>
<td>Lamp-MITL-6</td>
<td>S</td>
<td>TO</td>
<td>TO</td>
<td>594</td>
<td>1427</td>
</tr>
<tr>
<td></td>
<td>Lamp-MITL-7</td>
<td>U</td>
<td>246</td>
<td>308</td>
<td>235</td>
<td>244</td>
</tr>
</tbody>
</table>
### Table 6.7: Comparison between ae2zot and ae2bvzot on the Counting models.

<table>
<thead>
<tr>
<th>Model</th>
<th>R</th>
<th>ae2zot</th>
<th>ae2bvzot</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>T(s)</td>
<td>M(MB)</td>
</tr>
<tr>
<td>Counting1-1</td>
<td>S</td>
<td>142</td>
<td>347</td>
</tr>
<tr>
<td>Counting1-2</td>
<td>U</td>
<td>249</td>
<td>406</td>
</tr>
<tr>
<td>Counting1-3</td>
<td>U</td>
<td>1222</td>
<td>487</td>
</tr>
<tr>
<td>Counting1-4</td>
<td>S</td>
<td>204</td>
<td>363</td>
</tr>
<tr>
<td>Counting1-5</td>
<td>U</td>
<td>849</td>
<td>395</td>
</tr>
<tr>
<td>Counting2-1</td>
<td>S</td>
<td>8</td>
<td>271</td>
</tr>
<tr>
<td>Counting2-2</td>
<td>U</td>
<td>1487</td>
<td>291</td>
</tr>
<tr>
<td>Counting2-3</td>
<td>S</td>
<td>22</td>
<td>283</td>
</tr>
<tr>
<td>Counting2-4</td>
<td>S</td>
<td>4</td>
<td>258</td>
</tr>
<tr>
<td>Counting2-5</td>
<td>U</td>
<td>959</td>
<td>287</td>
</tr>
<tr>
<td>Counting2-6</td>
<td>S</td>
<td>3</td>
<td>247</td>
</tr>
<tr>
<td>Counting2-7</td>
<td>S</td>
<td>46</td>
<td>305</td>
</tr>
<tr>
<td>Counting2-8</td>
<td>-</td>
<td>TO</td>
<td>TO</td>
</tr>
<tr>
<td>Counting2-9</td>
<td>S</td>
<td>31</td>
<td>306</td>
</tr>
<tr>
<td>Counting2-10</td>
<td>S</td>
<td>9</td>
<td>274</td>
</tr>
<tr>
<td>Counting2-11</td>
<td>-</td>
<td>TO</td>
<td>TO</td>
</tr>
<tr>
<td>Counting2-12</td>
<td>S</td>
<td>23</td>
<td>276</td>
</tr>
<tr>
<td>Counting3-1</td>
<td>U</td>
<td>65</td>
<td>315</td>
</tr>
<tr>
<td>Counting3-2</td>
<td>U</td>
<td>577</td>
<td>379</td>
</tr>
</tbody>
</table>
Chapter 6. Tools and Experimental Evaluation

Table 6.8: Comparison between \textit{ae2zot} and \textit{ae2bvzot} on the Spikes and Wave models.

<table>
<thead>
<tr>
<th>Model</th>
<th>Tool</th>
<th>R</th>
<th>\textit{ae2zot}</th>
<th>\textit{ae2bvzot}</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>T(s)</td>
<td>M(MB)</td>
</tr>
<tr>
<td>Spikes-1</td>
<td>S</td>
<td>1</td>
<td>180</td>
<td>157</td>
</tr>
<tr>
<td>Spikes-2</td>
<td>S</td>
<td>13</td>
<td>311</td>
<td>293</td>
</tr>
<tr>
<td>Spikes-3</td>
<td>S</td>
<td>384</td>
<td>424</td>
<td>337</td>
</tr>
<tr>
<td>Spikes-4</td>
<td>U</td>
<td>554</td>
<td>654</td>
<td>403</td>
</tr>
<tr>
<td>Spikes-5</td>
<td>U</td>
<td>644</td>
<td>500</td>
<td>711</td>
</tr>
<tr>
<td>Spikes-6</td>
<td>U</td>
<td>983</td>
<td>520</td>
<td>757</td>
</tr>
<tr>
<td>Spikes-7</td>
<td>U</td>
<td>2920</td>
<td>498</td>
<td>375</td>
</tr>
<tr>
<td>Spikes-8</td>
<td>U</td>
<td>3132</td>
<td>517</td>
<td>TO</td>
</tr>
<tr>
<td>Spikes-9</td>
<td>S</td>
<td>614</td>
<td>477</td>
<td>1120</td>
</tr>
<tr>
<td>Spikes-10</td>
<td>S</td>
<td>702</td>
<td>439</td>
<td>393</td>
</tr>
<tr>
<td>Spikes-11</td>
<td>S</td>
<td>16</td>
<td>318</td>
<td>5</td>
</tr>
<tr>
<td>Wave-1</td>
<td>S</td>
<td>22</td>
<td>249</td>
<td>195</td>
</tr>
<tr>
<td>Wave-2</td>
<td>S</td>
<td>47</td>
<td>297</td>
<td>229</td>
</tr>
<tr>
<td>Wave-3</td>
<td>S</td>
<td>45</td>
<td>283</td>
<td>201</td>
</tr>
<tr>
<td>Wave-4</td>
<td>S</td>
<td>323</td>
<td>290</td>
<td>221</td>
</tr>
<tr>
<td>Wave-5</td>
<td>S</td>
<td>20</td>
<td>246</td>
<td>198</td>
</tr>
<tr>
<td>Wave-6</td>
<td>S</td>
<td>145</td>
<td>329</td>
<td>247</td>
</tr>
<tr>
<td>Wave-7</td>
<td>S</td>
<td>267</td>
<td>406</td>
<td>267</td>
</tr>
<tr>
<td>Wave-8</td>
<td>S</td>
<td>141</td>
<td>350</td>
<td>250</td>
</tr>
<tr>
<td>Wave-9</td>
<td>S</td>
<td>681</td>
<td>368</td>
<td>267</td>
</tr>
<tr>
<td>Wave-10</td>
<td>S</td>
<td>137</td>
<td>330</td>
<td>248</td>
</tr>
</tbody>
</table>
Conclusion

One of the concerns of software systems is that they are usually huge and complex, and there is a need to guarantee that the system does what we expect to do. There are software development methodologies like MDE that are used to reduce the systems complexity and pave the way for efficient software development by abstracting and modeling. Formal verification has a long track record in guaranteeing compliance of the system with its requirements. The necessity of verification, specially formal verification, rises from the fact that unexpected behavior of many systems like safety critical systems, malfunction of a component may cause irretrievable damages to the system itself and its environment that includes humans.

UML is used that provides a significant set of diagrams to describe structural and behavioral aspects of the system. UML models, as artifacts of the design phase, demonstrate the prospective software system at an abstract level. Focusing on the models instead of ultimate software artifact, raises the feasibility and efficiency of formal verification techniques, by hampering amplifying and propagating model defects. Therefore, it is favorable to start the formal verification process as soon as possible, in an incremental manner, on the models.

This chapter concludes our journey through scalable formal verification of UML models by reviewing the research questions introduced in Chapter 1 and describing how they are addressed in this thesis. Then a set of perspectives for future work is defined.

7.1 Reviewing Research Questions

Let us review the research questions, and discuss the contribution of this thesis towards them.

Most of the work has been done in ascribing formal semantics to UML concentrate
only on some diagram types, while the most unexpected and critical errors are due to inconsistencies among different diagrams. Different UML diagrams, like Class Diagram (CD), Object Diagram (OD), Interaction Overview Diagram (IOD), Sequence Diagram (SD), and State Machine (SM) have been formalized using several formalisms, such that the formal model is very similar to the original model, and there is a convenient way of mapping elements in the model to the corresponding elements in the formal model. For example, Label Transition System (LTS) is used for state machines and Petri net for activity diagrams.

Different diagram types focus on different aspects of the system. For example state machines monitor states of important objects in the system, while, sequence diagrams capture the behavior of objects interacting with one another in order to fulfill the functionality of the system. Supporting a set of heterogeneous behavioral views is desirable to meet industry requirements, like software life-cycle, documentation and code generation. For the sake of performing a comprehensive formal verification, all these views must be regarded, ideally notwithstanding how an organization handles its models.

**RQ 1.** How to formally specify diverse UML diagrams as a single integrated model that captures all possible behaviors of a system?

As a semi-formal language, UML has a formal syntax and an informally defined semantics. In fact, its semantics is intentionally underspecified to provide leeway for domain-specific refinements. On the other hand, general semantics hampers formal verification owing to the lack of required precision. The behavior of the modeling elements must be stated unambiguously, whereas informal semantics of UML diagrams result in different interpretations. These ambiguities mainly rise from sequence diagrams, which are often used to capture the most significant scenarios that describe how the components of a complex system interact. Focusing on sequence diagrams, many researchers propose their interpretations, among which only few [37, 53] present formalization amenable for formal verification. Although, their formalizations are based on fixed semantics with limited interoperability.

**RQ 1.1.** How to formalize those UML elements with many meanings, without losing generality?

The first contribution [7, 70] is a flexible modular formalization based on temporal logic for eFT-UML elements with many meanings, that mainly belong to SD. We studied the most significant semantics proposals, organized them into a single coherent framework, and proposed a solution to interpret SD in a compositional and modular way, in order to fulfill OMG’s ambition of keeping UML useful in many domains. Users can decide the interpretations of the key aspects of their interest and the result is a complete and coherent semantics; then, provided some simple constraints are respected to avoid making inconsistent decisions, our framework accommodates all other aspects.

The proposed theoretical approach is implemented on top of our verification toolset, namely Corretto\(^1\), where the produced temporal logic formula is fed to Zot\(^2\) our bounded model/satisfiability checker that allows the user to easily play with the different semantics. The user can simulate the behavior of the diverse semantics and verify the

---

\(^1\) https://github.com/deib-polimi/Corretto/
\(^2\) https://github.com/fm-polimi/zot
satisfiability of the properties of interest. S/he can thus understand how the system behaves or work on the satisfiability of the properties and then obtain the guarantees the system must offer. The same theoretical approach can be used for SM.

The next step towards formal verification is to use a verification technique to verify the formally specified model, while the state explosion has always been a crucial problem. With respect to the problem under consideration, where the formal verification of UML models is reduced to satisfiability checking of LTL formulae, as the UML model becomes more complex, the resulting LTL formula grows in size accordingly and makes the whole verification process slow, and at a given point impossible.

RQ 2. How to formally verify bigger UML models or generally any model specified in LTL?

While working on a solution for making the verification of UML models scale properly, we have identified an interesting solution that suites a wider class of verification problems that can be reduced to TRIO, therefore Constraint LTL (CLTL)/LTL formulae. The second contribution of this thesis [8, 70] is a bit-vector-based encoding of these CLTL/LTL formulae, which has allowed us to move a significant step forward in tackling the scalability of their formal verification. The proposed encoding for LTL and CLTL are implemented as plugins entitled bvzot and ae2bvzot in Zot our satisfiability/model checker.

7.2 Limitations and Future Work

The approaches described in this thesis suffer from a number of threats and weaknesses and suggest a number of improvement and future direction in both theoretical and technical aspects. We discuss some of them in this section and define the future work.

The proposed formalization of SD covers a the most frequently used and expressive CFs, which are alt, opt, loop, par, break. There are other CFs to be explored and formalized, that are strict, critical, assert, ignore, consider, and neg. Even though they are not widely used. Their formalization is straightforward using our modular semantics presented in Chapter 4.

Currently, all the SDs follow one configuration out of six, while one might be interested to have a set of SDs with a specific configuration and another set with different configuration. It is a technical issue that can be easily implemented not only for SDs, but also for every single CF. In other words, our flexible modular formalization allows every CF to have a meaning independently of the meaning of other parts, based on the chosen configuration. Apart from missing CFs, employing same theoretical approach of SDs for SMs is a part of our short term future plan.

A criticism to Corretto is that a basic knowledge about temporal logic is required for designers to insert their properties of interest to be verified. Currently, designers can insert the property using a Timed Property Diagram or directly as a TRIO formula in Zot syntax. An appropriate translation of the properties from natural language or a list of suggested patterns is an interesting research direction that can be pursued with respect to this issue.

Regarding tackling the scalability problem at the level of formalization, a possible direction is to abstract away those parts of the model that are irrelevant to the property being checked. For example, if a property is related to a specific scenario, we can skip
Chapter 7. Conclusion

those SDs whose objects do not engage in the scenario being investigated. However, a theoretical approach is required to identify the UML elements irrelevant to the property and guarantee that omitting them or their abstraction does not lead to false positive or false negative verification result.

Focusing on the scalability issues the level of temporal logic encodings, we are trying to directly translate TRIO formulae to a formula that is a combination of QF_BV, QF_LIA, and QF_LRA. Currently, TRIO formula is first translated to LTL, or CLTL if there are arithmetic constraints, and then we use either of two bit-vector-based encoding presented in this thesis. We expect that it improve the efficiency of satisfiability checking by excluding possible translation overhead and encode the TRIO temporal operators directly.

As pointed out in the experimental evaluation of CLTL encoding in Section 6.5, the configuration of SMT solvers highly affects the efficiency of the satisfiability checking process. Choosing the most efficient configuration out of the many possible ones was a trial-and-error process guided by our intuition of what reasonable tactics could be. We do not claim that this process led us to the absolute best possible configuration of the SMT solver for checking CLTL/CLTLoc specifications. We are studying the tactics available in z3 SMT solver to find better configurations with respect to the shape of the SMT problems that are produced by the bounded decision procedure for CLTL/CLTLoc.

We plan to investigate the possibility of exploiting a recent evolution of the NuSMV model checker, called nuXmv [21, 65] as the basis to implement the decision procedures for CLTL and CLTLoc. In fact, although nuXmv per se cannot handle precisely CLTL and CLTLoc models because it does not natively introduce certain conditions that are necessary for the decision procedures developed in [10,14], we aim to use it as an engine to develop further, novel techniques for solving such models.

We would also like to further stress the scalability issue by investigating the use of distributed model checking and proper abstractions on top of the smart encoding. Distributed model checking helps speed up the verification through the concurrent verification of pieces of the resulting model, then properly composed at the end. Abstractions and slicing can help find the minimal formalization/model that is precise enough to analyze a particular property and guarantee the correctness of obtained results. The last phase of the work will apply the different analysis techniques on a significant benchmark of old and new UML models to assess and evaluate their characteristics and draw some final guidelines.
Appendices
In this following, the detailed formalization of different CFs in TRIO formulae controlled by Java-like pseudo-code is presented. The overall formalization of a CF is the conjunction of all the formulae defined for corresponding CF. For every CF of the type $X$, $CF_X$ is the function that produces the TRIO formula, with respect to the given configuration, that is an object with respect to Figure 4.9, and $UML_{CF_X}$ that is the corresponding UML object.

### A.1 CF_Alt

This section presents formal semantics for CF_Alt assuming that, it has $m$ OPs, possibly an extra OP namely OPElse, $m$ Boolean expressions, each of which guards corresponding OP, and $n$ involved lifelines.

```plaintext
CF_Alt(UML_CF_Alt, config)
1. Borders(CF_Alt, SD_Stop)
2. Link_Pre_Post(CF_Alt, config)
Having $UML_{CF_Alt}$, we produce semantics of module CF_Alt and link it with the preceding and the following events on each lifeline.
3. Order(CF_AltStart, CF_AltEnd, True, SD_Stop, True)
Order between start and end event of CF_Alt is defined, which has no guard (the 3rd parameter is True), and events are allowed to be concurrent (the last parameter is True). This rule can be violated if and only if, we have SD_Stop between the occurrence of two events.
4. if (config.combine == WS){
5.  $\bigvee_{i=1}^{n} CF_Alt_LiStart \Rightarrow CF_Alt$
```
Appendix A. Appendix

6. $\text{CF}_\text{Alt}_{\text{Start}} \Rightarrow \bigvee_{i=1}^{m} \text{CF}_\text{Alt}_L_{i,\text{Start}}$

The sentence “$\text{CF}_\text{Alt}$ gets activated as soon as one lifeline enters.” can be implemented as conjunction of two sentences: “Start point of all involved lifelines are inside the border of $\text{CF}_\text{Alt}$.” (#5) and “Start point of $\text{CF}_\text{Alt}$ is concurrent at least with the start point of one lifeline.” (#6).

7. $\bigwedge_{i=1}^{m} \bigwedge_{j=1}^{n} \text{Order} \left( \text{CF}_\text{Alt}_L_{i,\text{Start}}, \text{CF}_\text{Alt}_P_{i,\text{Start}}, \text{CF}_\text{Alt}_P_{i}, \text{SD}_{\text{Stop}}, \text{True} \right)$

The start point of $\text{CF}_\text{Alt}_L_j$ is bidirectionally linked to the start point of $\text{CF}_\text{Alt}_P_{i,\text{Start}}$, provided that $\text{OP}_{i}$ is allowed to get activated. In other words, as soon as first lifeline enters and chooses one $\text{OP}_{i}$, and forces other lifelines to follow its footsteps.

8. $\bigwedge_{i=1}^{m} \bigwedge_{j=1}^{n} \text{OrderMonoD} \left( \text{CF}_\text{Alt}_P_{i,\text{End}}, \text{CF}_\text{Alt}_L_{j,\text{End}}, \text{True}, \text{SD}_{\text{Stop}}, \text{True} \right)$

There are mono-directional links from end points of $\text{OP}$ lifelines to the end points of $\text{CF}_\text{Alt}$ lifelines, in order to make sure that all possible ways of each lifeline, through different $\text{OP}$s, will join at the end point of $\text{CF}_\text{Alt}$’s corresponding lifeline.

Axioms created by #7 and #8 are also repeated for $\text{OPElse}$ in order to control orders.

9. if $\text{CF}_\text{Alt}$ has an Else operand{

10. $\text{CF}_\text{Alt}_O\text{PElse}_{\text{Start}} \iff \left( \neg \left( \bigvee_{i=1}^{m} \text{CF}_\text{Alt}_G_{i} \right) \land \text{CF}_\text{Alt}_{\text{Start}} \right)$

Else OP of $\text{CF}_\text{Alt}$ starts, if and only if all $m$ guards are evaluated as False at start point of $\text{CF}$.

11. if (config.Choice == ND)

12. $\text{CF}_\text{Alt}_{\text{Start}} \Rightarrow \left( \bigvee_{i=1}^{m} \left( \text{CF}_\text{Alt}_P_{i,\text{Start}} \land \neg \left( \bigvee_{j=1}^{m} \text{CF}_\text{Alt}_G_{j} \right) \right) \lor \text{CF}_\text{Alt}_O\text{PElse}_{\text{Start}} \right)$

At the start point of $\text{CF}$, either OPElse starts or an OP, while other OPs do not. This axiom captures the behavior of choosing one true guarded OP nondeterministically, that is sufficient condition for starting OP. Each OP also has a necessary condition (#22).

13. (\text{CF}_\text{Alt}_O\text{PElse}_{\text{End}} \lor \bigvee_{i=1}^{m} \text{CF}_\text{Alt}_P_{i,\text{End}}) \iff \text{CF}_\text{Alt}_{\text{End}} \}

CF ends at the moment the chosen OP ends (when all the involved lifelines leave the OP).

14. if $\text{CF}_\text{Alt}$ does not have an Else operand{

15. (\text{CF}_\text{Alt}_{\text{Start}} \land \neg \bigvee_{i=1}^{m} \text{CF}_\text{Alt}_G_{i} ))

At the moment the first lifeline enters the guards are evaluated and if all of them evaluated to False, all lifelines will collapse, therefore $\text{CF}_\text{Alt}$ will be ignored.

16. if (config.choice == ND)

17. (\text{CF}_\text{Alt}_{\text{Start}} \land \bigvee_{i=1}^{m} \text{CF}_\text{Alt}_G_{i}) \Rightarrow \bigvee_{i=1}^{m} \left( \text{CF}_\text{Alt}_P_{i,\text{Start}} \land \neg \bigvee_{j=1}^{m} \text{CF}_\text{Alt}_P_{j,\text{Start}} \right)$

If one or more guards are evaluated to true, one true guarded OP will be chosen nondeterministically.

18. $\bigvee_{i=1}^{m} \text{CF}_\text{Alt}_P_{i,\text{End}} \Rightarrow \text{CF}_\text{Alt}_{\text{End}} \}

19. if (config.choice == FFT)

20. $\bigwedge_{i=1}^{m} \left( \text{CF}_\text{Alt}_P_{i,\text{Start}} \iff \left( \text{CF}_\text{Alt}_{\text{Start}} \land \text{CF}_\text{Alt}_G_{i} \right) \land \neg \bigvee_{j=1}^{m} \text{CF}_\text{Alt}_G_{j} \right)$

This axiom guarantees that first true guarded OP from the top will be chosen. It says $i$th OP starts if and only if, its guard is evaluated to true and all guards belonging to previous OPs are evaluated to false.

21. if (config.choice == ND)

94
A.2. CF_Opt

22. \[ \bigwedge_{i=1}^{m} \left( \text{CF_Alt_OP}_i \Rightarrow (\text{CF_Alt}_\text{Start} \land \text{CF_Alt}_\text{Guard}_i) \right) \]
Imposes necessary condition for OP\(i\) to get started.
23. if (config.combine == SYNC) {
24. if CF_Alt has an Else operand {
25. \( \text{CF_Alt}_\text{ElseStart} \Leftrightarrow (\neg \bigvee_{i=1}^{m} \text{Guard}_i \land \text{CF_Alt}_\text{Start}) \)
26. if (config.choice == ND) {
27. \( \text{CF_Alt}_\text{Start} \Rightarrow (\bigvee_{i=1}^{m} (\text{CF_Alt}_\text{OP}_i \land \neg \bigvee_{j \neq i}^{m} \text{Guard}_j \land \text{CF_Alt}_\text{OP}_j \text{Start})) \land \bigvee_{i=1}^{m} (\text{CF_Alt}_\text{Else} \leftrightarrow \text{CF_Alt}_\text{ElseStart}) \)
28. \( (\text{CF_Alt}_\text{OPElseEnd} \lor \bigvee_{i=1}^{m} \text{CF_Alt}_\text{OP}_i \text{End}) \Rightarrow \text{CF_Alt}_\text{End} \)
29. if CF_Alt does not have an Else operand {
30. \( \text{CF_Alt}_\text{Start} \land \bigvee_{i=1}^{m} \text{CF_Alt}_\text{Guard}_i \Rightarrow \bigvee_{i=1}^{m} (\text{CF_Alt}_\text{OP}_i \text{Start}) \land \neg \bigvee_{j \neq i}^{m} \text{CF_Alt}_\text{OP}_j \text{Start}) \)
31. \( (\text{CF_Alt}_\text{Start} \land \neg \bigvee_{i=1}^{m} \text{Guard}_i) \lor \bigvee_{i=1}^{m} \text{CF_Alt}_\text{OP}_i \text{End}) \Rightarrow \text{CF_Alt}_\text{End} \)
32. if (config.choice == ND) {
33. \( \text{CF_Alt}_\text{Start} \Rightarrow (\bigvee_{i=1}^{m} \text{CF_Alt}_\text{OP}_i \text{Start} \land \neg \bigvee_{j \neq i}^{m} \text{Guard}_j) \lor \text{CF_Alt}_\text{End}) \)
34. if (config.choice == FFT) {
35. \( \text{CF_Alt}_\text{Start} \Rightarrow (\bigvee_{i=1}^{m} \text{CF_Alt}_\text{OP}_i \text{Start} \lor \text{CF_Alt}_\text{End}) \)
When CF_Alt starts either it collapses or one of its OPs gets started.
36. if (config.choice == ND) {
37. \( \bigwedge_{i=1}^{m} (\text{CF_Alt}_\text{OP}_i \text{Start} \Rightarrow \text{CF_Alt}_\text{Guard}_i \land \text{CF_Alt}_\text{Start}) \)
38. if (config.choice == FFT) {
39. \( \bigwedge_{i=1}^{m} (\text{CF_Alt}_\text{OP}_i \leftrightarrow (\text{CF_Alt}_\text{Start} \land \text{CF_Alt}_\text{Guard}_i \land \neg \bigvee_{j=1}^{i-1} \text{CF_Alt}_\text{Guard}_j)) \land \bigwedge_{i=1}^{m} \text{Combine}(\text{CF_Alt}_\text{OP}_i, \text{config}) \land \text{Combine}(\text{CF_Alt}_\text{OPElse}, \text{config}) \)
Finally, we pass the UML element of OPs to Combine modular semantics, in order to translate their content. However, every OP may have some basic interactions, and CFs, just like SD does. This procedure of translating continues recursively to inner most CF.

A.2. CF_Opt

This section presents the formal semantics of CF_Opt that has one OP and possibly one guard.

CF_Opt(UML_CF_Opt, config) {
1. Borders(CF_Opt, SD\text{\text{Stop}})
2. Link_Pre_Post(CF_Opt, config)
3. Order(CF_Opt\text{\text{Start}}, CF_Opt\text{\text{End}}, True, SD\text{\text{Stop}}, True)
4. if (config.combine == WS) {
5. \( \bigvee_{i=1}^{n} (\text{CF_Opt}_\text{L}_i \text{Start}) \Rightarrow \text{CF_Opt} \)
6. \( \text{CF_Opt}_\text{Start} \Rightarrow \bigvee_{i=1}^{n} (\text{CF_Opt}_\text{L}_i \text{Start}) \)
Appendix A. Appendix

7. $\bigwedge_{i=1}^{n} \text{Order}(\text{CF\_Opt\_Li}_\text{Start}, \text{CF\_Opt\_OP\_Li}_\text{Start}, \text{CF\_Opt\_OP}, \text{SD\_Stop}, \text{True})$
   The start point of CF\_Opt\_Li is bidirectionally linked to the start point of CF\_Opt\_OP\_Li, provided that OP is allowed to get activated. In other words, as soon as the first lifeline enters, it decides whether to enter into OP or ignore CF\_Opt and force other lifelines to follow its footstep.

8. $\bigwedge_{i=1}^{n} \text{OrderMonoD}(\text{CF\_Opt\_Op\_Li}_\text{End}, \text{CF\_Opt\_Li}_\text{End}, \text{True}, \text{SD\_Stop}, \text{True})$

9. $(\text{CF\_Opt}_\text{Start} \land \neg \text{CF\_Opt\_Guard}) \Rightarrow \bigwedge_{i=1}^{n} \text{SomFIn}_i((\text{CF\_Opt\_Li}_\text{Start} \land \text{CF\_Opt\_Li}_\text{End}), \text{CF\_Opt})$

10. $(\text{CF\_Opt\_OP}_\text{Start} \land \text{CF\_Opt\_Guard}) \Rightarrow (\text{CF\_Opt\_OP}_\text{Start} \lor (~\text{CF\_Opt\_OP}_\text{Start} \land \bigdot{\bigwedge}_{i=1}^{n} \text{SomFIn}_i((\text{CF\_Opt\_Li}_\text{Start} \land \text{CF\_Opt\_Li}_\text{End}), \text{CF\_Opt})))$
   At the start point of CF\_Opt if the guard is evaluated to true, the choice between activating (entering to CF\_Opt\_OP) and ignoring CF\_Opt will be made nondeterministically.

11. $(\text{CF\_Opt\_OP}_\text{End}) \Rightarrow \text{CF\_Opt}_\text{End}$

12. $(\text{CF\_Opt}_\text{End}) \Rightarrow (\bigdot{\bigvee}_{i=1}^{n} \text{CF\_Opt\_Li}_\text{End} \land \bigdot{\bigwedge}_{i=1}^{n} \text{SomPIn}_i((\text{CF\_Opt\_Li}_\text{End}), \text{CF\_Opt}))$

13. $(\text{CF\_Opt\_OP}_\text{Start} \land \text{CF\_Opt\_Guard}) \Rightarrow (\text{CF\_Opt\_OP}_\text{Start} \land \text{CF\_Opt}_\text{End})$

14. if (config\_combine == SYNCH) {
    15. $(\text{CF\_Opt}_\text{Start} \land \neg \bigdot{\bigvee}_{i=1}^{m} \text{CF\_Par\_Guard}_i) \Rightarrow \text{CF\_Par}_\text{End}$

16. $(\text{CF\_Par}_\text{Start} \land \neg \text{CF\_Par\_Guard}) \Rightarrow \text{CF\_Par}_\text{End}$

17. $(\text{CF\_Par}_\text{End}) \Rightarrow (\text{CF\_Par}_\text{Start} \lor \text{CF\_Par\_OP}_\text{End})$
   If CF\_Par is ignored, its start and end points coincide, otherwise the end point of CF\_Par coincides with the end point of CF\_Par\_OP.

18. $(\text{CF\_Par\_OP}_\text{Start}) \Rightarrow (\text{CF\_Par}_\text{Start} \land \text{CF\_Par\_Guard})$

19. $(\text{CF\_Par\_OP}_\text{End}) \Rightarrow \text{CF\_Par}_\text{End}$

20. Combine(CF\_Par\_OP, config)}

A.3 CF\_Par

This section presents the formal semantics for CF\_Par that has several OPs possibly guarded with Boolean expressions.

CF\_Par(UML\_CF\_Par, config) {
1-8 are same as the 1-8 in CF\_Alt.
9. $(\text{CF\_Par}_\text{Start} \land \neg \bigdot{\bigvee}_{i=1}^{m} \text{CF\_Par\_Guard}_i) \Rightarrow \bigwedge_{i=1}^{n} \text{SomFIn}_i((\text{CF\_Par\_Li}_\text{Start} \land \text{CF\_Par\_Li}_\text{End}), \text{CF\_Par})$
   If all the guards are evaluated to false, CF\_Par will be ignored by collapsing all involved lifelines.

10. $\bigwedge_{i=1}^{n} (\text{CF\_Par\_Start}_i \land \text{CF\_Par\_Guard}_i \leftrightarrow \text{CF\_Par\_OP}_i\text{Start}_i)$
   CF\_Par\_OP\_i starts iff, its guard is evaluated to true at the start point of CF\_Par.

11. $\bigwedge_{i=1}^{n} (\text{CF\_Par\_Li}_\text{End}_i \Rightarrow (\text{CF\_Par}_\text{Start} \land \text{SomPIn}_i((\text{CF\_Par}_\text{Start} \land \neg \bigdot{\bigvee}_{j=1}^{m} \text{CF\_Par\_Guard}_j), \text{CF\_Par})) \lor (\bigdot{\bigvee}_{j=1}^{m} \text{CF\_Par\_OP}_j\text{Li}_\text{End}_j \land \bigdot{\bigwedge}_{j=1}^{m} \text{SomPIn}_i((\text{CF\_Par\_OP}_j \Rightarrow \text{CF\_Par\_OP}_j\text{Li}_\text{End}_j), \text{CF\_Par}))}$

96
A.4. CF_Loop

CF_Par_Li ends if at the start point of CF all the guards are evaluated to false, or all true guarded OPs already finished their processing their events on Li.

12. \( \text{CF}_\text{Par}\_\text{End} \Rightarrow (\bigvee_{i=1}^{n}\text{CF}_\text{Par}\_\text{Li}_\text{End} \land \bigwedge_{i=1}^{n}\text{SomPIn}(\text{CF}_\text{Par}\_\text{Li}_\text{End}, \text{CF}_\text{Par})) \)

CF ends at the moment the last lifeline leaves. In other words, all the lifelines finished sometime in the past (including the current time instant) and the end point of CF is concurrent at least with one of them.

13. if (config.combine == SYNC) {
14. (CF_Par\_\text{Start} \land \neg\bigvee_{i=1}^{n}\text{CF}_\text{Par}\_\text{Guard}_i) \Rightarrow \text{CF}_\text{Par}\_\text{End} \\
15. \text{CF}_\text{Par}\_\text{End} \Rightarrow ((\bigvee_{i=1}^{n}\text{CF}_\text{Par}\_\text{OPi}_\text{End} \lor \text{CF}_\text{Par}\_\text{Start}) \land \bigwedge_{i=1}^{m}(\text{SomPIn}(\text{CF}_\text{Par}\_\text{OPi}_\text{End}, \text{CF}_\text{Par}) \lor \neg\text{SomPIn}(\text{CF}_\text{Par}\_\text{OPi}_\text{Start}, \text{CF}_\text{Par})))

The end point of CF_Par must be concurrent with its start point when getting ignored, and all of the OPs need to be accomplished or not started at all.

16. \( \bigwedge_{i=1}^{m}((\text{CF}_\text{Par}\_\text{Guard}_i \land \text{CF}_\text{Par}\_\text{Start}) \Leftrightarrow \text{CF}_\text{Par}\_\text{OPi}_\text{Start})) \\
17. \bigwedge_{i=1}^{m}\text{Combine}(\text{CF}_\text{Par}\_\text{OPi}, \text{config})

A.4 CF_Loop

The CF_Loop with min and max number of iterations has same behavior as min number of CF_Seqs followed by (max - min) number of CF_Opts, where all the CFs contain one OP equivalent to CF_Loop_OP. We have one UML element for CF_Loop_OP, which can get activated repeatedly. In WS configuration, we use the mechanism of simulating by putting sequence of CF_Seq and CF_Opt. Since lifelines can be in different iteration, we need to have different sets of predicates and map them to the UML elements inside CF_Loop_OP for each iteration. However, in the SYNC loop, one set of predicates for is sufficient, because no event is allowed to be interleaved with other events in different iterations, therefore, it can be controlled by a counter.

\[
\text{CF}_\text{Loop}(\text{UML}_\text{CF}_\text{Loop}, \text{config})\
\]

1-3 are same as the ones in CF_Alt.

4. if (config.loop == WS) {
5. \( \bigvee_{i=1}^{n}\text{CF}_\text{Loop}\_\text{Li}_\text{Start} \Rightarrow \text{CF}_\text{Loop} \\
6. \text{CF}_\text{Loop}_\text{Start} \Rightarrow \bigvee_{i=1}^{n}\text{CF}_\text{Loop}\_\text{Li}_\text{Start} \\
\text{CF}_\text{Loop} \text{ gets activated as soon as one lifeline enters.}
7. if (min > 0) {

If min is greater than zero, the first iteration on every lifeline will start by activating the start points of CF_Loop_OPi on every lifeline.

8. \( \bigwedge_{i=1}^{n}(\text{CF}_\text{Loop}\_\text{Li}_\text{Start} \Leftrightarrow \text{CF}_\text{Loop}\_\text{OPi}_\text{Start}) \\
9. \bigwedge_{i=1}^{n}\text{OrderMonoD}(\text{CF}_\text{Loop}\_\text{OPmin}_\text{Li}_\text{End}, \text{CF}_\text{Loop}\_\text{Li}_\text{End}, \text{True}, \text{SD}_\text{Stop}, \text{True}) \\
10. \bigwedge_{i=1}^{n}(\neg\text{CF}_\text{Loop}\_\text{Li}_\text{Skip} \Rightarrow \text{SomPIn}(\text{CF}_\text{Loop}\_\text{OPmin}_\text{Li}_\text{End}, \text{CF}_\text{Loop}\_\text{Li}))

We make sure that every involved lifeline has iterated at least min times, if there has not been any active CF_Break inside CF_Loop to ignore the whole loop. Note that CF_Loop\_Li\_Skip means that because of one CF_Break (Section A.5) the rest of events in the current iteration and all next iterations, therefore whole CF_Loop, on L_i, need to be
Appendix A. Appendix

ignored, and we have CF_Loop_Lj_End at the same time. Similarly, CF_Loop_OPi_Lj_Skip means that the rest of events in i'th iteration (OPi) need to be ignored and Lj jumps to the next iteration (OPi+1).

11. \( \bigwedge_{i=1}^{\text{max}} \bigwedge_{j=1}^{n} \text{CF_Loop_OPi}_Lj_{\text{Start}} \Rightarrow \neg \text{CF_Loop_Lj_Skip} \)
The necessary condition for the start point of every iteration on Lj, is the absence of any active CF_Break that causes ignoring the whole CF_Loop.

12. \( \bigwedge_{i=1}^{\text{min} - 1} \bigwedge_{j=1}^{n} (\text{CF_Loop_OPi}_Lj_{\text{End}} \Rightarrow (\text{CF_Loop_Lj_End} \lor \text{Next}(\text{CF_Loop_OPi+1}_Lj_{\text{Start}}))) \)
Mandatory iterations (1 to min) starts one time instant after previous iteration’s end point, except when whole CF_Loop ends.

13. \( \bigwedge_{i=1}^{\text{min} - 1} \bigwedge_{j=1}^{n} (\text{CF_Loop_OPi+1}_Lj_{\text{Start}} \Rightarrow \text{Yesterday}(\text{CF_Loop_OPi}_Lj_{\text{End}})) \)

14. for (i = min + 1; i <= max; i++){
15. CF_Loop_OPiStart => CF_Loop_Guard

Necessary condition for optional iterations is that guard is evaluated to true at their start points.

16. \( \bigwedge_{j=1}^{n} \text{OrderMonoD}(\text{CF_Loop_OPi}_Lj_{\text{End}}, \text{CF_Loop_Lj_End}, \text{True}, \text{SDStop}, \text{True}) \)
The end points of every optional iteration is mono-directionally linked to the end points of CF_Loop point on every involved lifeline. However, these are the possible ways to reach the end of CF_Loop, but not the only ones.

17. if (i == 1) {
18. \( \bigwedge_{j=1}^{n} \text{Order}(\text{CF_Loop_LjStart}, \text{CF_Loop_OPi}_Lj_{\text{Start}}, \text{CF_Loop_OPi}, \text{SDStop}, \text{True}) \)
When i is equal to one, it means that min is zero, accordingly there is not any mandatory iteration, and OPj’s lifelines are allowed to start provided that OPj holds at that time instant.

19. \( \text{CF_LoopStart} \land \neg \text{CF_Loop_Guard} \Rightarrow \bigwedge_{j=1}^{n} \text{SomFin}(\text{CF_Loop_LjStart} \land \text{CF_Loop_Lj_End}), \text{CF_Loop} \)
20. \( \text{CF_LoopStart} \land \text{CF_Loop_Guard} \Rightarrow (\text{CF_Loop_OPiStart} \lor (\neg \text{CF_Loop_OPiStart} \land \bigwedge_{j=1}^{n} \text{SomFin}((\text{CF_Loop_LjStart} \land \text{CF_Loop_Lj_End}), \text{CF_Loop})))) \}
Generally, for optional iterations, the guard is evaluated at their start points. If it is false, the whole loop is ignored, otherwise the choice between starting the next iteration and leaving loop is made nondeterministically.

21. if (i > 1) {
22. \( \bigwedge_{j=1}^{n} \text{Order}(\text{CF_Loop_OPi-i}_Lj_{\text{End}}, \text{CF_Loop_OPi}_Lj_{\text{Start}}, \text{Next}(\text{CF_Loop_OPi}), \text{CF_Loop_OPi}), (\text{CF_Loop_OPi} \lor \text{SDStop}), \text{False}) \)
The end point of iteration (i-1) on every involved lifeline is linked to the start points of next iteration (i), provided that CF_Loop_OPi holds at the next time instant when lifeline is done with the iteration (i-1). The last parameter of Order modular semantics is False, that means on each involved lifeline the start point of an iteration cannot be concurrent with the end point of previous iteration, and should be separated from each other at least by one time instant.

23. \( \bigwedge_{j=1}^{n} ((\text{CF_Loop_OPi-i}_Lj_{\text{End}} \land \text{Yesterday}(\neg \text{CF_Loop_OPi}) \land \neg \text{CF_Loop_Guard} \Rightarrow \neg \text{SomFin}(\text{CF_Loop_OPi}, \text{CF_Loop})) \)

98
A.4. CF_Loop

24. \( \bigwedge_{j=1}^{n} (\text{CF}_L_{\text{op}}_{i-1} \land \text{End}_{i}) \land \text{Yesterday}(-\text{CF}_L_{\text{op}}_{i}) \land \text{CF}_L_{\text{Guard}}) \)
\( \Rightarrow \) \( (\text{Next}(-\text{CF}_L_{\text{op}}_{i}) \lor \text{CF}_L_{\text{Guard}})) \)}

When a lifeline reaches the end point of an iteration, it evaluates the guard, provided that no other lifeline already initiated the next iteration. If the guard is evaluated to false, it makes sure that the next iteration will not get initiated in the loop (#23). Otherwise, the choice between starting the next iteration and leaving CF_Loop is made nondeterministically (#24).

25. \( \bigwedge_{j=1}^{n} (\text{CF}_L_{\text{end}}) \Rightarrow (\bigvee_{i=1}^{n} \text{SomPIn}(\text{CF}_L_{i}, \text{CF}_L)) \)

CF_Loop ends when last lifeline leaves it.

26. \( \bigwedge_{i=1}^{\max} \text{Combine}(\text{CF}_L_{\text{op}}_{i}, \text{config}) \)

It produces semantics for the content of each iteration. Note that skeleton and all borders are controlled in this modular semantics (CF_Loop).

27. \( \text{if} \text{ config.loop == SYNC} \{ \)

The semantics of SYNC loop is straightforward. Since iterations are separated by one time instant and not interfering one another, a set of predicates is enough to handle all the events. Accordingly, there is only one OP and one set of predicates mapped to it and its content.

28. \( \text{CF}_L_{\text{start}} \Rightarrow (\text{CF}_L_{\text{c}} = 0) \)

The counter is initiated by zero at the moment CF_Loop gets activated.

29. \( \text{CF}_L_{\text{start}} \land (\text{CF}_L_{\text{min}} = 0) \land \neg \text{CF}_L_{\text{guard}} \)
\( \Rightarrow (\neg \text{CF}_L_{\text{op}}_{\text{start}} \land \text{CF}_L_{\text{end}}) \)

30. \( \text{CF}_L_{\text{start}} \land (\text{CF}_L_{\text{min}} = 0) \land \text{CF}_L_{\text{guard}} \)
\( \Rightarrow (\text{CF}_L_{\text{op}}_{\text{start}} \lor (\neg \text{CF}_L_{\text{op}}_{\text{start}} \land \text{CF}_L_{\text{end}})) \)

It is possible that the loop has only optional iterations \( (\text{min} = 0) \). That being the case, the guard is evaluated at the start point of CF_Loop. If it is false, OP does not start and the whole CF_Loop collapses into one time instant (#29). Otherwise, the choice between starting OP and collapsing the whole CF_Loop is made nondeterministically (#30).

31. \( \text{CF}_L_{\text{start}} \land (\text{CF}_L_{\text{min}} > 0) \Rightarrow \text{CF}_L_{\text{op}}_{\text{start}} \)

First mandatory iteration starts unconditionally at the CF_Loop’s start point.

32. \( (\text{CF}_L_{\text{op}}_{\text{end}} \land (\neg \text{CF}_L_{\text{op}}_{\text{end}}) \)
\( \Rightarrow (\text{Next}(\text{CF}_L_{\text{c}}) = (\text{CF}_L_{\text{c}} + 1)) \)

At the end point of every iteration, except the last one, that is concurrent with the end point of CF_Loop, the counter is increased by one.

33. \( \neg (\text{CF}_L_{\text{op}}_{\text{end}} \lor \text{CF}_L_{\text{end}}) \)
\( \Rightarrow (\text{Next}(\text{CF}_L_{\text{c}}) = \text{CF}_L_{\text{end}}) \)

It is possible that the loop has only optional iterations \( (\text{min} = 0) \). That being the case, the guard is evaluated at the start point of CF_Loop. If it is false, OP does not start and the whole CF_Loop collapses into one time instant (#29). Otherwise, the choice between starting OP and collapsing the whole CF_Loop is made nondeterministically (#30).

34. \( (\text{CF}_L_{\text{op}}_{\text{end}} \land (\text{CF}_L_{\text{c}} + 1) < \text{CF}_L_{\text{min}}) \land (\neg \text{CF}_L_{\text{skip}}) \Rightarrow \text{Next}(\text{CF}_L_{\text{op}}_{\text{start}}) \)

Next mandatory iteration starts, with one time instant distance from previous iteration, provided that there is no active CF_Break that is ignoring the whole CF_Loop.

35. \( (\text{CF}_L_{\text{op}}_{\text{end}} \land ((\text{CF}_L_{\text{c}} + 1) \geq \text{CF}_L_{\text{min}}) \)

Next mandatory iteration starts, with one time instant distance from previous iteration, provided that there is no active CF_Break that is ignoring the whole CF_Loop.
Appendix A. Appendix

\( \land \neg CF\_Loop\_Guard \Rightarrow CF\_Loop\_End \)

36. \((CF\_Loop\_OP\_End \land ((CF\_Loop\_C + 1) \geq CF\_Loop\_Min) \land CF\_Loop\_Guard) \Rightarrow (Next(CF\_Loop\_OP\_Start) \oplus CF\_Loop\_End) \)

Only for the optional iterations, the guard is evaluated. If it is evaluated to false, CF\_Loop ends (#35), otherwise there is an exclusive choice between starting next iteration and ending CF\_Loop (#36).

37. \((if\ max\ !=\ "*"))

38. \((CF\_Loop\_OP\_End \land ((CF\_Loop\_C + 1) = CF\_Loop\_Max)) \Rightarrow CF\_Loop\_End \)

If it is not an infinite loop, this axiom is added, in order to control the upper bound of the number of iterations.

39. Combine\((CF\_Loop\_OP,\ config)\)}

A.5 CF\_Break

CF\_Break has only one OP and if its guard is evaluated to true after processing all the events inside OP rest of enclosing fragment will be ignored.

For CF\_Break, we propose new semantics which allows CF\_Break to break more than one border. It has additional field in its InteractionConstraint namely JumpLength, that determines how many borders CF\_Break must break. Suppose that there is a CF\_Break inside a CF\_Loop. When JumpLength is one, rest of iteration gets ignored and next iteration starts (similar to continue in programming languages). When it is two, the rest of iteration, and whole CF\_Loop is ignored. When it is three, the rest of iteration, whole CF\_Loop and its enclosing fragment are ignored, and so on for higher values. However, when JumpLength is one, CF\_Break's semantics is completely compliant with OMG specification.

\[ CF\_Break(UML\_CF\_Break,\ config) \{ \]

1-8 are similar to the ones in CF\_Alt.

9. \( \land_{i=1}^{n} \)Order\((CF\_Break\_Li\_Start, CF\_Break\_OP\_Li\_Start, CF\_Break\_OP, SD\_Stop, True) \)

10. \( \land_{i=1}^{n} \)OrderMonoD\((CF\_Break\_OP\_Li\_End, CF\_Break\_Li\_End, True, SD\_Stop, True) \)

11. \((CF\_Break\_Start \land \neg CF\_Break\_Guard) \Rightarrow \land_{i=1}^{n} SomFIni((CF\_Break\_Li\_Start \land CF\_Break\_Li\_End), CF\_Break) \)

When the guard is evaluated to false, this axiom makes sure that all involved lifelines collapse or will collapse, and accordingly CF\_Break completely gets ignored.

12. \((CF\_Break\_Start \land CF\_Break\_Guard) \Rightarrow CF\_Break\_OP\_Start \)

Being concurrent with the start point of CF\_Break and having the guard evaluated to true is the sufficient condition for OP to start.

13. \( \land_{i=1}^{n} ((CF\_Break\_OP\_Li\_End \land CF\_Break\_OP\_Li\_Start) \Rightarrow Next(CF\_Break\_Li\_End)) \)

14. \( \land_{i=1}^{n} (CF\_Break\_OP\_Li\_End \land \neg CF\_Break\_OP\_Li\_Start) \Rightarrow CF\_Break\_Li\_End \)
A.5. CF_Break

Since no element is allowed to start when the skip predicates of its lifelines hold, the start and end points of CF_Break (also the start and end points of its lifelines) should not be concurrent, even if CF_Break is empty. For example, when we have an empty CF_Break as the first element of another CF, we may have both skip and start of a lifeline at the same time instant, which is a contradiction. These axioms (#13 and #14) take care of this separation.

15. CF_Break_End \Rightarrow (\bigvee_{i=1}^{n} CF_Break_LiEnd \land \\
\land_{i=1}^{n} SomPIn(CF_Break_LiEnd, CF_Break))

16. CF_Break_OP_Start \Rightarrow (CF_Break_Start \land CF_Break_Guard)

17. \land_{i=1}^{n}((CF_Break_LiEnd \land (CF_Break_OP_LiEnd \lor Yesteraday)) \Rightarrow \land_{j=1}^{JumpLength} CF_Break_EFList_j_LiEnd)

By processing SD, we create a list out of enclosing fragments of CF_Break, CF_Break_EFList, whose elements end points can be triggered by current CF_Break. As soon as an active CF_Break ends, the end point of first JumpLength elements in the list are triggered. An active CF_Break ends, on an involved lifeline, when we have both CF_Break_End and CF_Break_OP_End, either simultaneously or with one time instant distance (for empty CF_Breaks).

18. for (i = 1; i <= JumpLength; i++)
19. for (j = 1; j <= n; j++)
20. CF_Break_EFList_i_Lj.SkipTriggers.add ((CF_Break_LiEnd \land (CF_Break_OP_LiEnd \lor Yesterday (CF_Break_OP_LiEnd))))

The end point of an active CF_Break on a lifeline is an event that sets skip and end predicate of some other CFs on the same lifeline. Every CF skips, on its lifelines, are triggered by disjunction of active CF_Breaks end point on the same lifeline.

21. if (config.combine == SYNC){
22. (CF_Break_Start \land \neg CF_Break_Guard) \Rightarrow CF_Break_End
23. (CF_Break_Start \land CF_Break_Guard) \Rightarrow CF_Break_OP_Start
24. CF_Break_End \Rightarrow ((CF_Break_Start \land \neg CF_Break_Guard) \lor Yesterday(CF_Break_OP_End \land CF_Break_OP_Start) \lor (CF_Break_OP_End \land \neg CF_Break_OP_Start))

CF_Break has three triggers

25. CF_Break_OP_Start \Rightarrow (CF_Break_Start \land CF_Break_Guard)
26. (CF_Break_OP_End \land CF_Break_OP_Start) \Rightarrow Next(CF_Break_End)
27. (CF_Break_OP_End \land \neg CF_Break_OP_Start) \Rightarrow CF_Break_End
28. \land_{i=1}^{n}((CF_Break_End \land (CF_Break_OP_LiEnd \lor Yesteraday)) \Rightarrow \land_{j=1}^{JumpLength} CF_Break_EFList_j_LiEnd)

for (i = 1; i <= JumpLength; i++)
29. for (j = 1; j <= n; j++)
30. CF_Break_EFList_i_Lj.SkipTriggers.add ((CF_Break_End \land (CF_Break_OP_LiEnd \lor Yesteraday(CF_Break_OP_LiEnd))))

31. Combine(CF_Break_Op, config)
Appendix A. Appendix

A.6 Combine

In the modular semantics of CFs, we produce semantics of CFs and skeleton of its OPs. The content of OPs are produced in turn by invoking Combine modular semantics. Inside each OP, there might be only basic interactions, and no need to invoke Combine. But, if an OP contains some CFs, the required modular semantics for those CFs are invoked, and consequently Combine modular semantics is invoked for their OPs. Finally, this exploration continues until the innermost CFs. This modular semantics is invoked for the elements in the first level of an SD or in the first level of an OP, in order to combine basic interactions and CFs.

**Combine**(UML\_Module, config){
1. Borders(Module, SD\textsubscript{Stop})
2. \(\bigwedge_{i=1}^{n}\)Borders(Module\_Li, SD\textsubscript{Stop})
3. if (UML\_Module is an SD)
   4. \(\bigwedge_{i=1}^{n}(SD\textsubscript{Start} \iff SD\textsubscript{L_iStart})\)
The start point of SD is synchronized with the start points of its lifelines.
5. if (UML\_Module is an OP){
   6. Module\_End \Rightarrow (\bigvee_{i=1}^{n} Module\_LiEnd \land \bigwedge_{i=1}^{n} SomPIn(Module\_LiEnd, Module))
The OP ends at the moment the last lifeline leaves it. Be noted that the start points of OP on each involved lifeline are controlled in the semantics of the enclosing CF.
7. \(\bigwedge_{i=1}^{n}\)Module\_Li \Rightarrow Module
   This axiom makes sure that the involved lifelines in the module are not active outside of the module borders.
8. if (config.combine == WS)
   9. Module\_Start \Rightarrow \bigvee_{i=1}^{n} Module\_LiStart
In the WS configuration, the start point of the module is required to be concurrent at least with the start point of one of its involved lifelines.
10. if (config.combine == SYNC)
The start point of OP is required to be concurrent with the start point of all of its involved lifelines in SYNC configuration.
11. Module\_Start \iff \bigwedge_{i=1}^{n} Module\_LiStart}
12. Order(Module\_Start, Module\_End, True, SD\textsubscript{Stop}, True)
13. \(\bigwedge_{i=1}^{n}\)Order(Module\_LiStart, Module\_LiEnd, True, SD\textsubscript{Stop}, True)
14. for (i = 1; i <= n; i++){
15. if (EV[i].Size == 0)
16. else{
17. if (EV[i][0] is an M)
18. Order(Module\_LiStart, EV[i][0], True, SD\textsubscript{Stop}, True)
19. EV[i][EV[i].Size - 1] \Rightarrow Module\_LiEnd
The start point of the module on Li is bidirectionally linked to the first event on Li. We have the end point of the module on Li at the moment the last event of Li occurs.
20. Module\_LiEnd \Rightarrow (EV[i][EV[i].Size - 1] \lor Module\_LiSkip)
A.7. Link_Pre_Post

The last event on \(L_i\) is not the only trigger of Module_\(L_i\_End\) and any active CF_Break inside the scope of the module can cause Module_\(L_i\_Skip\), therefore Module_\(L_i\_End\).

22. if ((\(EV[i]\).Size > 1) && (\(EV[i][0]\) is an M) && (\(EV[i][1]\) is an M))

23. \(\text{Order}(EV[i][0], EV[i][1], \text{True}, SD\_\text{Stop}, \text{True})\)

24. for (\(j = 1; j < EV[i].\text{Size} - 1; j++\))

25. if ((\(EV[i][j]\) is an M) && (\(EV[i][j + 1]\) is an M))

26. \(\text{Order}(EV[i][j], EV[i][j + 1], \text{True}, SD\_\text{Stop}, \text{True})\)

If the first two events are messages (Ms), first M is bidirectionally linked to the second M (#23). All other consecutive Ms are bidirectionally linked. Note that Link_Pre_Post modular semantics manages links between CF and other events.

27. Module_\(L_i\_End\) ⇔ \((\bigvee_{n_i = 1}^{n} \text{Module}_L_i\_End \land \bigwedge_{i=1}^{n} \text{SomPIn}_i(\text{Module}_L_i\_End, \text{Module}))\)

After adding these axioms, the whole Module is traversed for CFs, their modular semantics are invoked, and semantics are recursively produced here. For example, if we pass UML_SD and the configuration as the parameters to Combine modular semantics, the output is the semantics for the whole SD.

28. \(\bigwedge_{i=1}^{n} \text{Module}_L_i\_Skip \leftrightarrow \bigvee_{j=1}^{n} \text{Module}_L_i\_SkipTriggers\)

Any event in the Module_\(L_i\_SkipTriggers\) list (a list, which is incrementally created in CF_Breaks modular semantics) can trigger Module_\(L_i\_Skip\), therefore skip the rest of the events on \(L_i\) and give an end to \(L_i\) in scope of the module.

A.7. Link_Pre_Post

A module cannot trigger any event on a lifeline until its start point gets triggered by preceding event on that lifeline. Generally, the start and end points of a module can be considered as ports that are required to be linked to outer elements. The ports of OPs are handled in their CFs modular semantics and those for CFs are handled in Link_Pre_Post modular semantics.

\textbf{Link_Pre_Post}(UML\_CF\_X, config) {

1. if (config.combine == WS) {
2. \(\bigwedge_{i=1}^{n} \text{Borders}(CF\_X\_Li, SD\_\text{Stop})\)
3. for (\(i = 1; i <= n; i++\))
4. if (CF\_X\_Li\_Pre is an SD\_\text{Start} or an OP\_\text{Start})
5. \(\text{Order}(CF\_X\_Li\_Pre, CF\_X\_Li\_Start, \text{True}, SD\_\text{Stop}, \text{True})\)

When the CF is the first element of an SD or an OP on \(L_i\), CF\_X\_Li\_Pre is bidirectionally and unconditionally linked to CF’s start point on \(L_i\). Unconditionally, because when its enclosing fragment starts, it means that it already obtained the permission of getting started.

6. else
7. \(\text{Order}(CF\_X\_Li\_Pre, CF\_X\_Li\_Start, \neg\text{EnclosingF}_L_i\_Skip, SD\_\text{Stop}, \text{True})\)

Otherwise, they are linked provided that there is no active CF_Break that forces the CF to be skipped. If there is any active CF_Break, which aims the CF’s enclosing fragment, it triggers enclosing fragment’s skip on the lifeline that is being investigated.

8. if (CF\_X\_Li\_Post is an SD\_\text{End} or an OP\_\text{End})
9. \(\text{OrderMonoD}(CF\_X\_Li\_End, CF\_X\_Li\_Post, \text{True}, SD\_\text{Stop}, \text{True})\)

When the CF is last event of an SD or an OP on \(L_i\), there is no need to check for active
CF_Breaks. Because, even if there is some active CF_Break that need to skip this CF, they give an end to it, so does this axiom. Since the CF’s end point on Li is not the only trigger for its enclosing fragment’s end point on Li, CF_X_Li_End is mono-directionally linked to its following event.

10. \textbf{else}
11. \textbf{Order(CF_X_Li_End, CF_X_Li_Post, \neg EnclosingF_Li_Skip, SD_{Stop}, True)}}</p>

Otherwise, like #7, events occur in normal order when there is no EnclosingF_Li_Skip.

12. \textbf{if (config.combine == SYNC){}
Since in SYNC configuration involved lifelines enter into, and leave CF in a synchronized way, there is no need for start and end points on each lifeline, therefore CF_X_Start and CF_X_End suffice.

13. \textbf{for(i = 0; i < n; i++)}{
14. \textbf{if (CF_X_Li_Pre is an SDStart or an OPStart)\}
15. \textbf{Order(CF_X_Li_Pre, CF_X_Start, True, SD_{Stop}, True)\}
16. \textbf{else\}
17. \textbf{Order(CF_X_Li_Pre, CF_X_Start, \neg EnclosingF_Li_Skip, SD_{Stop}, True)\}
18. \textbf{if (CF_X_Li_Post is an SDEnd or an OPEnd)\}
19. \textbf{OrderMonoD(CF_X_End, CF_X_Li_Post, True, SD_{Stop}, True)\}
20. \textbf{else\}
21. \textbf{Order(CF_X_End, CF_X_Li_Post, \neg EnclosingF_Li_Skip, SD_{Stop}, True)}}</p>
Bibliography


<table>
<thead>
<tr>
<th>No.</th>
<th>Author(s)</th>
<th>Title</th>
<th>Journal/Conference</th>
</tr>
</thead>
</table>
Bibliography


