Geração de propriedades sobre programas Java a partir de objetivos de teste

Simone Hanazumi

TESE APRESENTADA
AO
INSTITUTO DE MATEMÁTICA E ESTATÍSTICA
DA
UNIVERSIDADE DE SÃO PAULO
PARA
OBTENÇÃO DO TÍTULO
DE
DOUTORA EM CIÊNCIAS

Programa: Ciência da Computação
Orientadora: Prof^a. Dr^a. Ana Cristina Vieira de Melo

Durante o desenvolvimento deste trabalho a autora recebeu auxílio financeiro da CAPES e FAPESP (Processos: 2011/01928-1, 2012/23767-2 e 2013/22317-6)

São Paulo, Outubro de 2015

Geração de propriedades sobre programas Java a partir de objetivos de teste

Esta versão da tese contém as correções e alterações sugeridas pela Comissão Julgadora durante a defesa da versão original do trabalho, realizada em 29/10/2015. Uma cópia da versão original está disponível no Instituto de Matemática e Estatística da Universidade de São Paulo.

Comissão Julgadora:

- Prof^{a}. Dr^{a}. Ana Cristina Vieira de Melo (orientadora) - IME-USP
- Prof. Dr. Flávio Soares Corrêa da Silva - IME-USP
- Prof. Dr. Ricardo Luis de Azevedo da Rocha - EP-USP
- Prof. Dr. Valdivino Alexandre de Santiago Júnior - INPE
- Prof^{a}. Dr^{a}. Patrícia Duarte de Lima Machado - UFCG

"Unless you try to do something beyond what you have already mastered you will never grow."

- Ralph Waldo Emerson

Acknowledgments

Words are not enough to express my gratitude to everyone who supported me during my Ph.D. work. I thank ***God*** *for blessing me with this unique opportunity to proceed with my research with so many great supporters.*

I would like to thank my advisor, ***Dr. Ana C. V. de Melo****, for her patience and guidance. She has inspired me to continue my research, and gave me the chance to fulfill my potential through this Ph.D. work. I also would like to thank the* ***Institute of Mathematics and Statistics of the University of São Paulo*** *for providing all the structure I needed to develop my research.*

I want to thank ***Dr. Corina S. Păsăreanu*** *for sharing her knowledge and experience with me during my research internship in the USA. Under her supervision, I could greatly improve my work with the tool Java PathFinder. I also want to thank the C****arnegie Mellon University - Silicon Valley*** *and the* ***NASA Ames Research Center*** *for providing such a rich environment for research development.*

I would like to thank the ***Brazilian Ministry of Research Education (CAPES)*** *and the* ***State of São Paulo Research Foundation (FAPESP)*** *for their financial support during the Ph.D. course. Thanks to their support, I could dedicate my whole effort to the development of this Ph.D. research.*

I also want to thank all the ***professors*** *and* ***colleagues*** *that I met during my Ph.D. course. Thank you for sharing your vision and for helping me to improve my knowledge and maturity.*

And a special thanks to ***my dearest family*** *and* ***my most valuable friends****. Many thanks for always being there for me and for having my back.*

Thank You

Abstract

HANAZUMI, S. **Generation of Java program properties from test purposes**. 2015. Thesis (Ph.D.) - Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2015.

The task of guaranteeing that computational systems do not fail and work correctly has become extremely important with the growing presence of new technologies in people's lives. Therefore, it is essential to ensure that such systems work properly to confirm their high-quality and to avoid financial and even life losses.

One of the techniques used to this purpose is called *formal verification of programs*. From the system specification, which should be described in a formal language, we define properties that must be satisfied during system execution to guarantee the software quality. Then, these properties are checked using a verifier, which is the tool responsible for running the verification and for notifying whether the property was satisfied by the program; if the property was violated, it indicates to software developers the possible location of faults in the system. The disadvantages of using formal verification are the high cost to apply this technique in practice, and the necessity of having people with experience in formal methods to derive the properties from system specification and define them in a formal representation that can be read by a program verifier. This particular task of deriving a property from system specification and defining it to be checked by a verifier is complex, time-consuming and error-prone, since it is usually done by hand.

To help software developers in the application of formal verification in Java programs, we propose in this work the generation of properties formal representation for direct use in a verifier. The generated properties are test purposes, which are derived from system formal specification and present the desirable system behavior that must be observed during the system execution. Establishing that the universe of properties correspond to the universe of test purposes of a program, we guarantee that the generated properties describe the expected program behavior through execution traces that lead to either an *accept* state or a *refuse* state. Thus, when the verifier checks the test purpose, it can give a success/fail verdict for the property, and provide traces coverage data that can be used to analyze the program behavior that led to that verdict.

Keywords: Program Specification, Test Purposes, Java, Formal Verification.

Resumo

HANAZUMI, S. **Geração de propriedades sobre programas Java a partir de objetivos de teste**. 2015. Tese (Doutorado) - Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2015.

Com a presença cada vez maior de sistemas computacionais e novas tecnologias no cotidiano das pessoas, garantir que eles não falhem e funcionem corretamente tornou-se algo de extrema importância. Além de indicar a qualidade do sistema, assegurar seu bom funcionamento é essencial para se evitar perdas, desde financeiras até de vidas.

Uma das técnicas utilizadas para esta finalidade é a chamada *verificação formal de programas*. A partir da especificação do sistema, descrita numa linguagem formal, são definidas propriedades a serem satisfeitas e que certificariam a qualidade do software. Estas propriedades devem então ser implementadas para uso num verificador, que é a ferramenta responsável por executar a verificação e informar quais propriedades foram satisfeitas e quais não foram; no caso das propriedades terem sido violadas, o verificador deve indicar aos desenvolvedores os possíveis locais com código incorreto no sistema. A desvantagem do uso da verificação formal é, além do seu alto custo, a necessidade de haver pessoas com experiência em métodos formais para definir propriedades a partir da especificação formal do sistema, e convertê-las numa representação que possa ser entendida pelo verificador. Este processo de definição de propriedades é particularmente complexo, demorado e suscetível a erros, por ser feito em sua maior parte de forma manual.

Para auxiliar os desenvolvedores na utilização da verificação formal em programas escritos em Java, propomos neste trabalho a geração de representação de propriedades para uso direto num verificador. As propriedades a serem geradas são objetivos de teste derivados da especificação formal do sistema. Estes objetivos de teste descrevem o comportamento esperado do sistema que deve ser observado durante sua execução. Ao estabelecer que o universo de propriedades corresponde ao universo de objetivos de teste do programa, garantimos que as propriedades geradas em nosso trabalho descrevem o comportamento esperado do programa por meio de caminhos de execução que levam a um estado de aceitação da propriedade, ou a um estado de violação. Assim, quando o verificador checa o objetivo de teste, ele consegue dar como resultado o veredicto de sucesso ou falha para a propriedade verificada, além de dados da cobertura dos caminhos de execução do programa que podem ser usados para análise do comportamento do programa que levou ao sucesso ou falha da propriedade verificada.

Palavras-chave: Especificação de Programas, Objetivos de Teste, Java, Verificação Formal.

Contents

List of Abbreviations

DFA	Deterministic Finite Automaton
GUI	Graphic User Interface
IDE	Integrated Development Environment
IUT	Implementation Under Test
JFLAP	Java Formal Language and Automata Package
JPF	Java PathFinder
LTL	Linear Temporal Logic
NFA	Nondeterministic Finite Automaton
SP	Safety-Progress
SPS	Specification Pattern System

List of Figures

List of Tables

I— Work Overview and Background

1 — Introduction

This chapter introduces the motivation and the main objectives of this work. First, we describe what motivated us, which is the complex task of applying formal verification techniques in systems written in the Java programming language. An example of the problem we want to deal with is also presented with the motivation, to clarify the difficulties that can arise when facing it. Next, we describe the aim of this research, which is to help developers in the process of applying model checking in their Java programs, and the research main objectives. A brief discussion concerning to related work follows the research objectives section. Finally, at the end of the chapter, we present the organization of this thesis.

1.1 Motivation

The advents of the Internet and new technologies have increased the presence of computational systems in our daily life. For each bank transaction, or even to make a simple phone call, there is a system that makes it possible to accomplish these operations. And, since we are more and more dependent of such systems to accomplish daily tasks, there is an increasing concern in guaranteeing the proper operation of these programs. Particularly, companies of the technology area want to ensure the high-quality of their software since software faults can result in financial and even life losses [Pre01, BK08]. Hence, aiming to confirm the good quality of the developed software, one of the most promising approaches is the use of the systems formal verification.

The formal verification technique intends to guarantee that programs are running according to a specification, or are accomplishing a set of important properties. Nowadays, there is an effort in the development of tools that help in the formal verification of programs, including the systems written in object oriented languages like Java, which is the language that we have chosen for this work [DPJ08, DRH06, VHB$^+$03, HDD$^+$03, RDH04, EM04, DHJ$^+$01, PDV01, HD01]. Ensuring the proper operation of programs through formal verification has a considerable impact in the total cost of software development.

One of the disadvantages of this technique is related to the feasibility of applying the formal verification during a system development. The use of formal verifiers (e.g. model checkers) depends on predefining properties, since many software developers do not have adequate training or experience to define new properties. And, even if there are developers that are able to accomplish the assignment of implementing new properties in a formal verifier, it is still a complex and time-consuming task that increases the final cost of the product as consequence.

Focusing at the application of formal verification in systems written in the Java language [GJSB05, Ora], the central idea of this work is to generate a formal representation of properties that should be satisfied by the system in a semi-automated manner. This representation is generated from test purposes, which can be derived from system formal specification. *Test purposes* (or *observation objectives*) [dT01] describe what is being tested concerning to a particular system specification requirement. It presents *the desirable system behavior that we want to observe during its execution*, describing the execution sequences that lead to an *accept* state and/or to a *refuse* state (Section 2.2). Hence, we consider as system properties the test purposes concerning to the system specification (Section 2.2.1). Establishing that the universe of properties correspond to the universe of test purposes of a program, we guarantee that the generated properties describe the expected program behavior through execution traces that lead to either an *accept* state or a *refuse* state. Thus, when we input the test purpose representation to the model checker, it can give a success/fail verdict for the property, and traces coverage data that can be used to analyze the program behavior that led to that verdict. Therefore, the aim of this research is to simplify the formal representation of properties and its use along with a model checker, to verify Java systems in a more efficient and faster manner.

Remark 1. As we stated earlier, we consider in our research system properties that are test purposes. Test purposes (or observation objectives) are always related to the specified program behavior that we want to observe during testing/verification execution [dT01]. Moreover, test purposes can be represented using automata that describe the program paths that lead to an accept or a refuse state. Thus, when a test purpose is refused by a program, we can track the program execution paths that led to that verdict.

⇒ Details regarding test purposes (observation objectives) and properties can be viewed in Sections 2.2 and 2.2.1.

Remark 2. In this research we have chosen to work with the Java programming language due to the worldwide use of Java and its features. Java libraries comprise a wide range of computational areas (e.g. concurrent programming and distributed computing), and it has been used in several areas, from embedded systems to web applications. Therefore, the programs that are written in Java compose a rich set of system that can be used in the evaluation of this research proposed approach for properties definition and formal verification of programs.

1.1.1 Approaches for property generation and verification

Considering the test purposes we want to check in a system, we derive properties that should be specified in a formal manner. To generate the specification of these properties in a semi-automated manner, guaranteeing that the generated specification can be by a model checker, we need to have a good understanding of properties formal specification and the mechanisms of model checking. For instance, consider the following Java code:

Listing 1.1: *Example - Java Code*

```java
public class Example {
        public static void aMethod (){
                System.out.println("A Method");
        }
        public static void otherMethod (){
                System.out.println("Other Method");
        }
        public static void main(String[] args) {
                boolean condition = true;
                if (condition) aMethod();
                otherMethod();
        }
}
```

Suppose that in the specification of the system, we have the test purpose: *"the method `aMethod` is called at least once during program execution"*. The test purpose specify that program execution paths with at least one occurrence of `aMethod` lead the test purpose to an *accept state*; otherwise, the test purpose reaches a *refuse state*, which means that it was violated by the program. Figure 1.1 presents the automaton representation of the test purpose. *The refuse state has a thick dashed border, while the accept state has a thick plain border.*

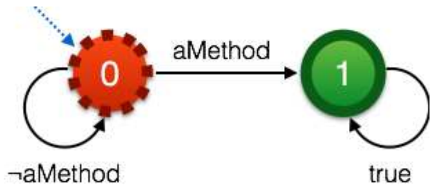


Figure 1.1: *Example: test purpose automaton representation*

According to the literature [BK08, Hol97, VHB$^+$03], we can use one of two approaches to specify a test purpose (or property) and verify it with a model checker:

1. Implement the formally specified properties in a model checker using the system programming language (in our case, Java), and then use the model checker to verify the system code directly;
2. Model the system in a formal language so that we can define properties in a model checker directly at the specification level.

In the *first approach*, system properties are specified using a formal language, so we need to deal with the gap between specification and implementation levels to define the properties as they were specified. On the other hand, the *second approach* requires us to model the system implementation using another formal specification language. In addition, we may need to deal with different specification languages for the properties and for the system model (e.g. in the Spin model checker [Hol97], properties can be specified in linear temporal logic (LTL)[Pnu77], and the system must be modeled in Promela, which means that we have to deal with two formal specification languages, and guarantee that the Java system is modeled in Promela correctly). Both tasks are time-consuming, error-prone and complex, since we have to deal with specification and implementation levels. *However, since our focus is to simplify the application of model checking in Java systems, we chose the first approach.*

To apply the first approach in the specification of the property *"the method `aMethod` is called at least once during program execution"*, we need to implement the property by hand in the model checker, guaranteeing that the property implementation agrees with its specification. The implementation of the property depends on the chosen model checker, and the developer should deal with the gap between specification and code levels to guarantee that the property code behaves as expected by the specification. For instance, recall the property that we used in the example of Listing 1.1, which stated that the method *"public static void aMethod();"* is called at least once during the example execution. As we presented previously, we could specify the property using an automaton (Figure 1.1). But using the first approach, we must define the property at implementation level. For this purpose, suppose that we use the model checker Java PathFinder (JPF) [VHB$^+$03, Jav], which allows us to implement such property in Java. One way to define this property in JPF is by extending the *"Property Listener Adapter"* class (Listing 1.2). The property violation occurs when no execution of *"aMethod()"* is detected.

Listing 1.2: *Example - JPF Property Implementation*

```
public class ExampleProperty  extends PropertyListenerAdapter {
        private int count = 0;

```

```
	@Override
	public boolean check(Search search, JVM vm) {
		return (count > 0);
	}
	@Override
	public void instructionExecuted(JVM vm) {
		Instruction instruction = vm.getLastInstruction();
		MethodInfo mi = instruction.getMethodInfo();
		if (mi.getName().equals("aMethod")) {
			count ++;
		}
	}
}
```

As we can observe, to implement the property in Java, we must first analyze the Java code of the program (Listing 1.1) and its behavior. Then, we must define the JPF property (Listing 1.2) by hand, using the available resources of the model checker. In small examples this manual approach can be applied. However, when we have bigger Java codes, this approach may become unfeasible due to the increased complexity of the analysis of the program behavior, and to the increased time that is required to understand and learn all the model checker resources and restrictions to define a property as it was specified. Furthermore, guaranteeing that the property implementation agrees with its specification is a complex task. Hence, instead of analyzing the Java code (Listing 1.1) and then implement each property of the system specification by hand (which is a time-consuming and error-prone approach), we propose in this Doctorate the use of the following steps:

1. Convert the property specification in a formal representation that can be applied in a model checker for the system code (in our case, the model checker must be able to check Java programs);
2. Choose a model checker that can be used to read the formal representation of the property and then verify the system code directly. To make the model checker able to read the formal representation, we may need to make changes in the model checker code.

To define a methodology for the conversion of a property specification into a formal representation that will be read by a model checker, we need to have a deep understanding of the process of specifying a property formally (Chapter 2 and Part II). In addition, we need to understand the mechanisms of a Java model checker to certify that it will be able to read the converted property and verify it against the corresponding program (Part III). *But once we find a conversion methodology and make the adaptation of the chosen model checker to verify the converted property, we can simplify and generalize the process of representing a system property formally and perform the system formal verification* (Figure 1.2).

1.2 Objectives

The use of formal verification technique in the development of software is restricted since for a given system, we need to implement specific properties in a verifier to check its correctness. And the implementation of such properties is an expensive and time-consuming task, even if the software developer has a good knowledge of formal methods.

Focusing in simplifying the application of formal verification techniques in practice, we propose the generation – from a test purpose derived from the system formal specification – of properties formal representation for direct use in a verifier for Java programs (Figure 1.2). These properties should describe the ideal behavior of the system, according to its formal specification, so that the program verification can detect code faults if at least one of those properties is not satisfied. For instance, when we are dealing with a critical system, it would be interesting to verify if a property that describe the exception-handling procedures for a specific exception is satisfied; if the property

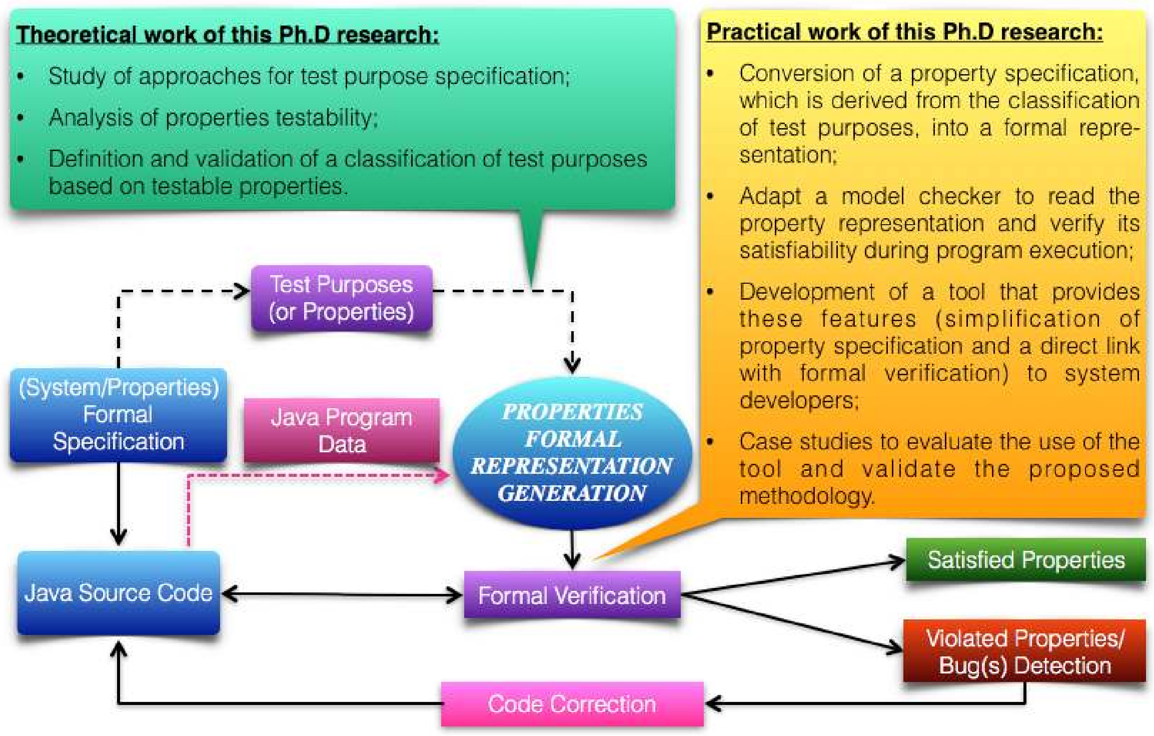

Theoretical work of this Ph.D research:
• Study of approaches for test purpose specification;
• Analysis of properties testability;
• Definition and validation of a classification of test purposes based on testable properties.
Practical work of this Ph.D research:
• Conversion of a property specification, which is derived from the classification of test purposes, into a formal representation;
• Adapt a model checker to read the property representation and verify its satisfiability during program execution;
• Development of a tool that provides these features (simplification of property specification and a direct link with formal verification) to system developers;
• Case studies to evaluate the use of the tool and validate the proposed methodology.
Test Purposes (or Properties)
(System/Properties) Formal Specification
Java Program Data
PROPERTIES FORMAL REPRESENTATION GENERATION
Java Source Code
Formal Verification
Satisfied Properties
Violated Properties/ Bug(s) Detection
Code Correction

Figure 1.2: *Ph.D research: overall approach.*

is not satisfied, this would mean that the exception was not handled accordingly and, therefore, it could result in the spread of even more serious errors in the system.

Due to the important results we can get from model checking application in Java programs, we target the simplification of the process of generating a formal representation of system properties (test purposes), and then use these representations in a Java model checker. Hence, the main objectives of this research are:

- **Objective 1.** Development of a methodology to convert properties that are derived from test purposes and specified in a formal language to a formal representation that could be used in a Java formal verifier. This objective can be divided in two parts:

  (a) ***Part I (Theoretical Work).*** This part involves the study of existing approaches for test purposes specification, and the definition of the universe of properties we should handle in our research. To this intent of defining a universe of properties, we must establish a connection between test purposes specification and properties testability. Then, a systematic way to specify a property, and then convert it into a formal representation that can be read by a model checker, must be defined.

  (b) ***Part II (Practical Work).*** This part connects the property specification and formal representation with model checking. A Java model checker must be adapted to read the property formal representation, and verify the property satisfiability during program execution. This part is responsible for providing the mechanisms we need to implement our own tool that covers property specification and formal verification.

- **Objective 2 *(Practical Work).*** Implementation of a tool that will make this feature (simplification of property specification and verification of the defined property with a model checker) available to software developers. This objective is part of the practical work of this research.

- **Objective 3 *(Practical Work).*** Evaluation of the tool through its application in a set of case studies in order to assess the validation of the developed methodology. The approach for this evaluation is not statistical, but it is focused in the analysis of the advantages and restrictions of our tool application in practice. This objective is part of the practical work of this research.

1.3 Related Work

In this section, we discuss some related work regarding our research. They are divided by topics that are covered in our work.

1.3.1 Integration of Formal Methods and System Development

To apply the formal verification technique, it is necessary to integrate formal methods to the system development. Gargantini et al. [GRS09] propose the combination of formal methods with the MDE (Model-Driven Engineering) paradigm to reduce the disadvantages and take the advantages that each part present separately. Möller et al. [MORW08, MORW04] have developed the CSP-OZ – a formal method that combines the CSP process algebra [Hoa85] with the specification language Object-Z [Smi00] – and integrates it with UML [Fow03] and Java to allow the construction of system models since the early stages of their development, making it possible to apply formal verifiers from the beginning of software projects. Another approach that aims at the usage of formal methods in the development of object-oriented programs code is proposed by Ahrendt et al. [ABB+00]. In their work, they aim to make the use of formal methods in systems development more feasible in order to apply formal methods techniques in industry.

1.3.2 System Modeling and Property Specification

Another usage of formal methods in software engineering is in system specification and modeling. Such usage make not only the development process easier, but also allow the application of formal verifiers to test and verify the resulting system in later stages. Long and Long [LL03] presented a Z [WD96] specification of a Java concurrent model in order to assist the use of formal verification in Java concurrent systems. Yang and Poppleton [YP07] also focused in the development of correct and reliable concurrent Java systems, and they have proposed the translation of B [Wor96] and CSP system specification to Java, being the specification verified by the ProB tool [LB03].

Regarding property specification, we have approaches that generate test purposes from a system implementation [dSM06, WFW09]. Since we are interested in deriving test purposes from system specification, and then apply the test purposes in the formal verification of programs, we proceed to the discussion of works that target the simplification of the process of property specification. The Specification Pattern System (SPS), which was proposed by Dwyer et al. [DAC99], can be used to reduce the complexity of properties specification. It defines scopes and general patterns that can be instantiated to specify a system property. The correctness of the SPS is discussed by Salamah et al. [SGRM05]: through an empirical study using the SPIN model checker [Hol97] and several test cases, they presented the validation conditions for the SPS.

Inspired by the SPS, other researchers created similar approaches for property specification. For instance, Konrad and Cheng [KC05b] proposed an adaptation of the SPS to the context of real time systems. Castillos et al. [CDJ+13] proposed a compositional automata based semantics for the SPS patterns, since the SPS lacks a compositional semantics. And Bauer et al. [BLS06] created SALT, a new specification language with a higher level of abstraction, when compared to other similar approaches. It incorporates SPS patterns, but provides nested scopes, exceptions, and offers support for regular expressions and real-time.

Aiming at the semi-automation of properties specification, tools have been developed. Prospec [MGR04] provides a user-friendly interface to define properties in LTL using the SPS and composite propositions, which are abstractions used to handle multiple events and conditions in the SPS formulas. Spider [KC05a] is a tool based on the SPS whose objective is to assist practitioners in the specification of properties using natural language.

1.3.3 Use of Formal Verification in Practice

Focusing on the programs formal verification issue, Jin [Jin07] proposes an approach for properties formal verification in sequential Java programs. In this work, the properties are specified using finite state automata, and, from these automata, rules are generated and applied in the given program in order to search any code violation. Abdelhalim et al. [ASST10] and Möller [Möl02], in turn, use CSP to specify systems and their properties to run the formal verification and prove the correctness of the developed product. Parízek [Par08] presents in his Doctorate thesis techniques to apply formal verification in real components using the Java PathFinder (JPF) tool [Jav]. And Malik et al. [MTL10] make use of UML models and formal verification to test a given system.

Concerning to the development of tools for property specification and verification, we have the following related work. Balasubramanian et al. [BPN+11] defined properties using contracts or automata, which are generated using the SPS and then linked to Java code and model checker Java PathFinder (JPF) [Jav] using Matlab/Simulink [Mat]. JPF-LTL [JL] extends JPF with LTL verification, but it works in the context of programs with infinite execution. Belinfante [Bel10] developed the tool JTorX, which checks whether a test purpose specification holds in a Java program implementation using the *ioco* relation. The test purposes are represented in LTS and the underlying theory of *ioco* relation can be found in [Tre08]. Jard and Jéron [JJ05] developed the tool TGV, which generates test cases from test purposes represented using LTS.

Remark. In our research, we also target the integration of system development with formal methods. To do this, we propose an approach of simplifying the model checking application in a Java system code. First, we derive properties from system specification using mechanisms that are available in literature (Chapter 2 and Part II). Then, the next step is to establish a conversion of the property to a formal representation that must be readable by a Java model checker (Part III). The last step is to adapt the model checker to verify the property, and analyze the verification result.

$\Rightarrow$ *A summary of the scientific contributions of our work is presented in Chapter 7.* Discussions concerning to the differences and similarities of our research with related work are presented in the conclusion of Chapters 2, 3, 4, 5, and 6.

1.4 Organization of the Thesis

The thesis is organized in parts and chapters. Each part is composed by chapters related to a topic of this Doctorate research. A short description of each chapter is provided next.

Part I: ***Work Overview and Background.*** *This part introduces the main motivation and objectives of the work, and presents a brief description of background concepts.*

- **Chapter 1:** brief introduction that describes the motivation of this work, its main objectives, related work, and the organization of this document;
- **Chapter 2:** description of basic concepts (e.g. formal verification, system properties and test purposes, property specification) that are used throughout the thesis.

Part II: ***Properties Testability.*** *This part contains the chapters regarding the theoretical part of this Doctorate research. It is recommended for readers with interest in linear temporal logic (LTL), formal specification of properties and its validation.*

- **Chapter 3:** description of the properties testability concept and presentation of the study that establishes a connection between this concept and the SPS formulas in LTL;

- **Chapter 4:** validation of the SPS formulas that are able to derive testable properties specification through the analysis of formulas using LTL semantics.

Part III: ***Properties Verification.*** *This part presents the practical part of the research, and therefore, it is recommended for those readers who are interested in Java programs and its verification through model checkers. The practical approach we used in our research integrates the theoretical concepts into a tool, JPF-SPS, which can be applied in Java properties specification and verification. The JPF-SPS has been developed to simplify these processes, providing an intuitive graphic user interface to specify properties, and a verification process that is triggered through the click of buttons. Hence, the JPF-SPS was built so that any developer – even those developers who do not have an expertise in these fields of specification, verification and/or quality assurance (QA) – can use the tool to specify and verify properties. The main requirement to use JPF-SPS is a good understanding of the Java program behavior that is going to be verified. A basic knowledge of automata, properties specification and verification should be useful to a faster learning of the JPF-SPS features.*

- **Chapter 5:** description of the approach used to verify the system properties, and presentation of tools and mechanism that were used to develop the tool JPF-SPS;
- **Chapter 6:** presentation of the implemented tool JPF-SPS and its features through a set of Java examples.

Part IV: ***Final Considerations.*** *This part concludes the thesis by presenting the main results and contributions of the research.*

- **Chapter 7:** summary of the results that were achieved, description of the scientific contributions and future work and directions.

Part V: ***Appendix and References.*** *Additional material and the references that were used in this research are presented in this part.*

- **Appendix A:** list of Simone Hanazumi's publications concerning to the studies of software testing, formal verification, properties specification, and scientific contributions of this research.
- **Appendix B:** detailed overview of LTL and its related concepts;
- **Appendix C:** validation of SPS formulas that are used in a related work, through the analysis of formulas using LTL semantics;
- **Appendix D:** presentation of the automata library of JPF-SPS.
- **Glossary:** list of concepts that appear throughout the thesis.
- **Bibliography:** list of references that are cited throughout the thesis.

2 — Background

This chapter gives a description of the main concepts regarding test purposes, software formal specification and verification, which compose the theoretical basis of our work. The concepts that are described here were extensively used in our research, and appear throughout the thesis.

2.1 Formal Verification

Systems verification checks if a software has been developed correctly or not [Boe81]. Formally, it can be defined as the process to prove that given a program P and a specification φ, P satisfies φ (i.e. program P is a model for specification φ). [Fra92, MSF06]. This kind of verification is also known as *a posteriori* system verification since the specification and the program are provided. Figure 2.1 illustrates this verification process.

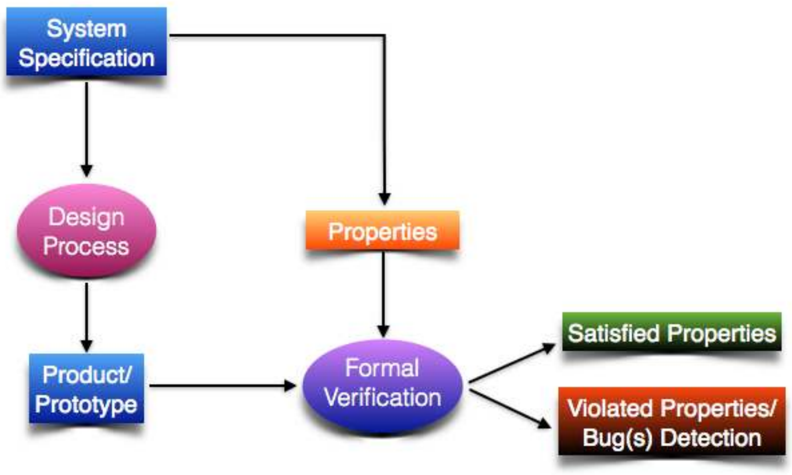


Figure 2.1: *Diagram of an* a posteriori *system verification adapted from [BK08].*

The steps of *a posteriori* system verification are described below:

1. *System Specification.* It is the basis of any verification activity, since it describes in a formal manner all features that should be implemented in the system. The formal description is necessary in order to have a precise specification of the system [MSF06].

2. *Properties Definition.* From system specification, one can derive properties that the system prototype must satisfy. An example of system property would be the absence of deadlocks in the system.

3. *Product/Prototype Development.* It refers to the implementation phase of the system prototype and/or product.

4. *System Verification.* Having the product ready for execution and properties already defined, we can run the system verification. This verification checks whether a property is violated or satisfied. If there is a property violation, a bug was found and the system is not correct according to its specification. In general, one should use tools that automate this verification process.

Formal methods and verification can guarantee a more reliable system implementation, however many people resist to the application of these methods in practice because they consider such methodology too complicated and complex. Besides, system formal specification is done by people and there is no guarantee that it agrees to what the system must do [Fra92].

2.1.1 Model Checking

Many approaches and techniques for systems verification were developed along the years. Particularly, in this research, we use the technique of model checking. In this technique, given a finite-state model of a system (e.g. a finite transition system) and a system property, the verification must explore all possible system model states in a brute-force manner to check if there is any violation of the property. If such violation exists, a counterexample that exposes the failure location is presented. Otherwise, it is concluded that the system model satisfied the property.

Properties that can be checked using model checking are of qualitative nature (e.g. generated results are equal to expected results) and/or related to timing (e.g. a system response is received within 5 minutes). Figure 2.2 presents a schematic view of model checking process.

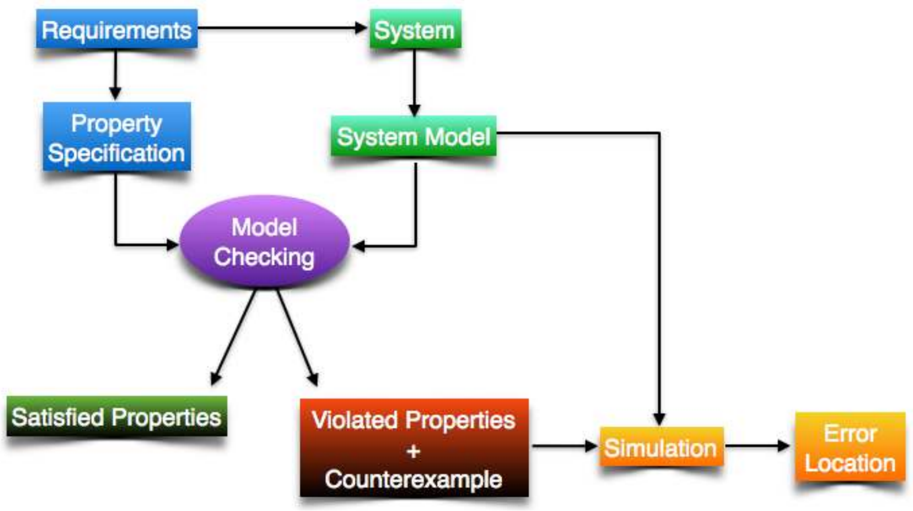


Figure 2.2: *Model checking diagram adapted from [BK08].*

To use model checking technique, the following phases must be accomplished [BK08]:

Modeling Phase: during this phase, a model of the system is created using a model checker description language. To assess the model, some simulations should be performed. Finally, a set of systems properties that must be satisfied should be specified using a property specification language.

Running Phase: in this phase, the model checker is executed to verify the soundness of the specified system properties in the system model.

Analysis Phase: during this phase, we analyze the verification results. If a property was satisfied, then we proceed to check the next property (if it exists). Otherwise, we must first analyze the counterexample that was generated by simulation; then, we should locate the error and correct it by refining the model design or property; finally, we repeat the entire procedure once again. If the system run out of memory, we must try to reduce the model and repeat the entire procedure with the new model.

Next, we present a formal definition of the model checking technique. It summarizes the statements we described above.

Definition 2.1.1 — Formal Definition of Model Checking. Suppose that we have a finite transition system TS that models a system, and a temporal logic formula φ that represents a system property. The model checking technique searches for a property violation in the system model, i.e., the model checking technique searches and detects the set of states (S) of TS that satisfies φ. Considering $s \in S$ a state of the model, we can define the model checking process as:

$$\{s \in S \mid (TS, s) \vDash \varphi\}$$

2.2 Conformance Testing and Test Purposes

The concept of test purposes appear in *conformance testing*, which is the activity of checking whether a system is correctly implemented according to its functional specification. Conformance testing involves black-box testing, therefore, the interaction with the system is done through external interfaces [dT01].

A *test purpose* or *observation objective*[1] [dT01, JJ05] is a description of what is being tested concerning to a particular system specification requirement. It presents the desirable system behavior that we want to observe during its execution, describing the execution sequences that lead to an *accept* state and/or to a *refuse* state. A test purpose can be used and described in several ways (e.g. automata [JJ05]). Here, we present the formal definition of a test purpose as stated by Tretmans [dT01]. The formal notion of a test purpose is given through the following definitions [dT01, dSM06]:

IUT. It is the **I**mplementation **U**nder **T**est or the system that is going to be tested.

IMPS. It is the class containing all IUT's.

MODS. It is the universe of implementation models. Here, we consider that each IUT can be modeled by a formal object i_{IUT} that belongs to the set $MODS$. Note that we only assume the existence of an i_{IUT}, but not that it is known.

SPECS. It is the universe of formal specifications.

TOBS. It is the universe of observation objectives.

Implementation Relation. It relates the implementation models and specifications: $imp \in MODS\ \mathcal{X}\ SPECS$. It is used to state that a model i_{IUT} conforms to the system specification.

Conformance. The conformance of a system implementation to its corresponding specification

[1]According to Tretmans [dT01], the term *test purpose* is too general and it is overloaded with many different meanings. Hence, to give a precise notion of a test purpose, he used the term *observation objective* instead. However, in the remaining of this work, we have chosen to keep the term *test purpose*.

is expressed by the relation: $\mathsf{conforms-to} \in \mathsf{IMPS} \, \mathcal{X} \, \mathsf{SPECS}$. For instance, if the IUT is a correct implementation of a specification $s \in \mathsf{SPECS}$, we say that IUT $\mathsf{conforms-to}$ s. Moreover, using the implementation relation and considering that the IUT can be modeled by $i_{IUT} \in \mathsf{MODS}$, we can state that IUT $\mathsf{conforms-to}\ s \Leftrightarrow i_{IUT}\ \mathsf{imp}\ s$.

Exhibition. An IUT exhibits an observation objective if it behaves as specified by the observation objective. This relation can be represented by $\mathsf{exhibits} \subseteq \mathsf{IMPS} \, \mathcal{X} \, \mathsf{TOBS}$.

Reveal Relation. Using the hypothesis that for all IUT there is a corresponding model in the universe of models ($\forall$ IUT $\in \mathsf{IMPS}\ \exists\ i_{IUT} \in \mathsf{MODS}$), we can define the reveal relation: $rev \subseteq \mathsf{MODS} \, \mathcal{X} \, \mathsf{TOBS}$. The reveal relation is used to decide which experiments should be done to challenge an IUT to exhibit the requested observation. It allows us to reason about exhibition in the formal domain.

Satisfying a Test Purpose. The IUT satisfies a test purpose $e \in \mathsf{TOBS}$ iff the model i_{IUT} is rev-related to e. Formally: IUT $\mathsf{exhibits}\ e \Leftrightarrow i_{IUT}\ rev\ e$.

Verdict Function. This function relates the observations obtained with testing experiments to a test purpose, showing the verdict if an evidence of an exhibition was seen. This verdict function is called the hit-function, and it is represented formally as: $H_e : \mathcal{P}(\mathsf{OBS}) \to \{\mathsf{hit}, \mathsf{miss}\}$, where $\mathcal{P}(\mathsf{OBS})$ is the set of observations. Here, we use hit if the verdict states that an exhibition evidence was found during the experiments; and miss otherwise.

Test Suite. Given a test suite T_e, we have the following definition: IUT $\mathsf{hits}\ e\ \mathsf{by}\ T_e =_{def} H_e(\bigcup\{\mathsf{exec}(t, \mathsf{IUT}) \mid t \in T_e\}) = \mathsf{hit}$, where $\mathsf{exec}(t, \mathsf{IUT})$ represents a test execution. A test suite can be:

- $e - \mathsf{complete}$: the test suite is able to distinguish among all exhibiting and non-exhibiting implementations. Formally: IUT $\mathsf{exhibits}\ e \Leftrightarrow$ IUT $\mathsf{hits}\ e\ \mathsf{by}\ T_e$.
- $e - \mathsf{exhaustive}$: the test suite is able to only detect non-exhibiting implementations. Formally: IUT $\mathsf{exhibits}\ e \Rightarrow$ IUT $\mathsf{hits}\ e\ \mathsf{by}\ T_e$
- $e - \mathsf{sound}$: the test suite is able to only detect exhibiting implementations. Formally: IUT $\mathsf{exhibits}\ e$ if IUT $\mathsf{hits}\ e\ \mathsf{by}\ T_e$

Inconclusive Verdict. An inconclusive verdict can occur when an IUT is correct but it does not exhibit the requested observation; or, when an incorrect IUT exhibits the test purpose.

2.2.1 Test Purposes versus Properties

An important remark is about the use of the term *property* and the term *test purpose* in our work. As we have stated previously, a test purpose is a description of a behavior that we must observe on an IUT. A property can be represented by a test purpose if it describes a program behavior either. However, we need to emphasize that there are properties that do not describe a system behavioral specification [FFJ+10].

In this work, our focus is on test purposes. Therefore, we consider only properties that can be represented by test purposes. We must also consider that properties can be represented by a temporal formula, since we work with the LTL specification language (besides regular expressions). As a result, if we have a property p represented by a test purpose e and a temporal formula φ, we assume that they are semantically equal, i.e., $p =_{sem.} e =_{sem.} \varphi$. *Thus, for simplicity, we can use the term property as a synonym of test purpose, and consider that a temporal formula can be used to represent both of them.* Hence, we can submit to a model checker a formally specified test purpose (property) to be verified against the IUT model.

2.3 Formal Specification Languages

This section presents formal description languages Linear Temporal Logic (LTL) and Regular Expressions. The LTL is a well known specification language in the field of formal verification, being widely used in model checking to represent system properties. Regular expressions are used to represent program events traces through sets of strings, and can be converted to other formal representations (e.g. deterministic finite automata (DFA)) in a direct manner. Due to their characteristics, both LTL and regular expressions are applied in our study to represent system properties. Their features and how they are used in our research are discussed in the next sections.

2.3.1 Java Program Example for Property Specification

Before we describe the formal specification languages, we present a simple Java program that will be used to illustrate the process of properties specification. This Java example (Listing 2.1) performs simple arithmetic operations over an integer x:

1. `add1`: this method adds one unit to the number x.
2. `reset`: this method sets the value of x to 0.
3. `isZero`: this method returns true if x has value 0, and false otherwise.

Listing 2.1: *Arithmetic Example - Partial Code*

```java
public class ArithmeticExample {
        public static int add1 (int x) {
                x = x + 1;
                return x;
        }
        public static int reset (int x) {
                x = 0;
                return x;
        }
        public static boolean isZero (int x) {
                return x == 0;
        }
        public static void main(String[] args) {
                ...
        }
}
```

Considering the above methods as program events, we can specify properties concerning to their occurrences and interactions throughout the program execution. The properties can be specified in several ways, but in our work, we are particularly interested in specifying properties using LTL (Section 2.3.2) and regular expressions (Section 2.3.3).

⇒ The properties specification process regarding the program `ArithmeticExample` is illustrated in Sections 2.3.2, 2.3.3, and 2.4.4.

2.3.2 LTL

The linear temporal logic (LTL) [Pnu77, MP91, MP92] has a broad application in model checking theory, and it can be used to represent programs events and states along time. Next, we give a brief overview of LTL operators and semantics according to the definitions presented in [BK08, MP91].

Definition 2.3.1 — Logic Operators. The unary logic operator for negation ($\neg$), and the binary logic operators for conjunction ($\bigwedge$), disjunction ($\bigvee$), implication ($\rightarrow$), equivalence ($\leftrightarrow$) and exclusive or ($\oplus$), can be applied to LTL formulas. Their usual meanings in propositional logics are preserved in LTL.

Definition 2.3.2 — Semantics Overview. Given a model σ and temporal formulas φ and γ, an inductive definition for the notion of a temporal formula φ holding at a position $j \geq 0$ in σ, denoted by $(\sigma, j) \models \varphi$ (satisfaction relation), is presented below.

$$
\begin{array}{lclcll}
(\sigma,j) & \models & \varphi & \Leftrightarrow & s_j \models \varphi,\ \text{i.e., } \varphi \text{ is evaluated locally using} & \\
 & & & & \text{the interpretation given by } s_j & \\
(\sigma,j) & \models & \neg\varphi & \Leftrightarrow & (\sigma,j) \not\models \varphi & \\
(\sigma,j) & \models & \varphi \vee \gamma & \Leftrightarrow & (\sigma,j) \models \varphi \text{ or } (\sigma,j) \models \gamma & \\
(\sigma,j) & \models & \bigcirc\varphi & \Leftrightarrow & (\sigma,j+1) \models \varphi & \text{Next operator} \\
(\sigma,j) & \models & \varphi\ \mathcal{U}\ \gamma & \Leftrightarrow & \text{for some } k \geq j, (\sigma,k) \models \gamma, & \text{Until operator} \\
 & & & & \text{and for every } i \text{ such that } j \leq i < k, (\sigma,i) \models \varphi & \\
(\sigma,j) & \models & \bullet\varphi & \Leftrightarrow & (\sigma,j-1) \models \varphi & \text{Previous operator} \\
(\sigma,j) & \models & \varphi\ \mathcal{S}\ \gamma & \Leftrightarrow & \text{for some } k \leq j, (\sigma,k) \models \gamma, & \text{Since operator} \\
 & & & & \text{and for every } i \text{ such that } j \geq i > k, (\sigma,i) \models \varphi &
\end{array}
$$

Definition 2.3.3 — Additional Temporal Operators. Additional LTL temporal operators can be defined using the operators that appeared in the *LTL semantics overview*.

$$
\begin{array}{lcll}
\Diamond\varphi & = & \text{True}\ \mathcal{U}\ \varphi & \text{Eventually operator} \\
\Box\varphi & = & \neg\Diamond\neg\varphi & \text{Always/Henceforth operator} \\
\varphi\ \mathcal{W}\ \gamma & = & \Box\varphi \vee (\varphi\ \mathcal{U}\ \gamma) & \text{Weak Until operator} \\
\blacklozenge\varphi & = & \text{True}\ \mathcal{S}\ \varphi & \text{Once operator} \\
\blacksquare\varphi & = & \neg\blacklozenge\neg\varphi & \text{Has-always-been operator} \\
\varphi\ \mathcal{B}\ \gamma & = & \blacksquare\varphi \vee (\varphi\ \mathcal{S}\ \gamma) & \text{Weak Since/Back-to operator}
\end{array}
$$

⇒ *A more detailed definition of LTL, its syntax and semantics, can be viewed in Appendix B.*

Specifying Properties in LTL

To illustrate the process of specifying a property in LTL, we use as an example properties concerning to the events of the program `ArithmeticExample` (Section 2.3.1). The events are: `add1`, `reset` and `isZero`.

■ **Example 2.3.1** Property: *"the method `add1` must be called at least once during the program execution"*. This property states that there is at least one call to the method `add1` for every program execution trace. Therefore, it can be specified in LTL using the eventually operator:

$$\Diamond\ \mathrm{add1}$$

■ **Example 2.3.2** Property: *"when the method `reset` is called, the method `isZero` is called in the future, i.e., a `reset` method call is followed by a call to `isZero`"*. It means that the occurrence of `reset` implies that `isZero` will eventually occur in the future. This can be specified in LTL using the implication and the eventually operators:

$$\mathrm{reset} \rightarrow \Diamond \mathrm{isZero}$$

2.3.3 Regular Expressions

According to Mitkov [Mit03], a regular expression describes a set of strings (i.e. a regular language) or a set of ordered pairs of strings (i.e. a regular relation). It can be easily converted to a finite-state automaton that represents the language or relation it describes. Table 2.2 presents the operators that are commonly used to write regular expressions [Mit03, Pat].

Definition 2.3.4 — Regular Expressions Events and Basic Symbols. Consider a regular expression over a finite alphabet of events Σ. The events of a regular expression can be represented using the symbols: $\emptyset$, ε, $e \in \Sigma$ and .. The description of each symbol is presented in Table 2.1.

Table 2.1: *Regular Expression Basic Symbols*

Symbol	*Description*
$\emptyset$	Empty event
ε	Empty string
e	Event e in alphabet Σ
.	Any event of the alphabet Σ

Definition 2.3.5 — Regular Expressions Operators. Assuming that $e_1, e_2, ...e_n$ are regular expressions, we can use them to generate new regular expressions by applying regular expression operators. Table 2.2 presents the set of regular expression operators that are used throughout this research thesis.

Table 2.2: *Regular Expression Operators*

Operator	*Description*
(e_1)*	Event e_1 occurs 0 or more times
$(e_1\|e_2)$	Event e_1 occurs or event e_2 occurs
$(e_1; e_2)$	Events e_1 and e_2 occur in sequence
$[-e_1]$	Any event can occur except e_1
$[-e_1, e_2, ..., e_n]$	Any event can occur except $e_1, e_2, ..., e_n$
$(e_1)?$	Event e_1 occurs 0 or 1 time, i.e., either e_1 occurs or ε occurs

Specifying Properties with Regular Expressions

To illustrate the process of specifying a property with regular expressions, we use as an example the same properties that were presented in Section 2.3.2. These properties concern to the events `add1`, `reset` and `isZero` of the program `ArithmeticExample` (Section 2.3.1).

■ **Example 2.3.3** Property: *"the method `add1` must be called at least once during the program execution"*. This property states that there is at least one call to the method `add1` for every program execution trace. We can specify this behavior using regular expression symbols and operators:

[−add1]*; add1; .*

Analyzing the above expression, it states that any event (except `add1`) can occur 0 or more times before the occurrence of `add1`; and, after we observed the occurrence of `add1`, any other event can occur 0 or more times.

■ **Example 2.3.4** Property: *"when the method `reset` is called, the method `isZero` is called in the future, i.e., a `reset` method call is followed by a call to `isZero`"*. It means that the occurrence of `reset` implies that `isZero` will eventually occur in the future. This behavior can be specified using regular expression operators as follows:

```
[−reset]*; (reset; [−isZero]*; isZero; [−reset]*)*
```

This expression states that first, any event (except `reset`) can occur 0 or more times. Then, the following pattern can occur 0 or more times: the method `reset` is followed by a future occurrence of `isZero`; any event may occur 0 or more times between `reset` and `isZero`, and the occurrence of `isZero` cannot be followed by 0 or more occurrences of `reset` that are not followed by other occurrences of `isZero`.

2.4 Specification Pattern System (SPS)

Specification patterns [DAC99, Pat] are parameterizable, high-level, formalism-independent specification abstractions defined for finite state verification. Their purpose is to assist practitioners in mapping descriptions of system behavior into their formalism of choice, improving the transition of these formal methods to practice. The patterns are divided in a few categories, and their instances can represent the majority of system specification properties. This result is interesting for our research, as this formal system restricts properties specification to a few patterns, but still preserves an expressiveness that allows the formal description of most of the system specification properties.

The next sections give a brief overview of this system. For a complete collection of specification patterns and their conversion to several specification languages, like LTL and regular expressions, please refer to [Pat].

2.4.1 Scopes

For each pattern, we must first define its *scope*. The *scope* is the extent of the program execution over which the pattern must hold, and is determined by specifying a starting and an ending state/event for the pattern. Dwyer et al. divided these scopes in five categories: global, before R, after L, between L and R, and, after L until R. L and R represent the states/events that limit the scope on the left-hand side and on the right-hand side respectively. A general description of each scope is presented next. It contains an informal definition of the scope – as defined by Dwyer et al. [DAC99, Pat] – and the scope characterization described by Salamah et al. [MGR04, SGRM05, SGRE11], which is used in our research to specify system properties.

Global

This scope comprises the entire program execution, including all the states in the computation.

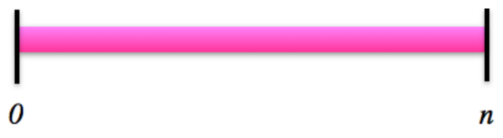


Figure 2.3: *Global Scope*

Global scope characteristics [SGRE11]

1. The scope denotes the entire computation;

2. The scope includes all the states in the computation;
3. The interval defined by the scope occurs once in a computation.

Before *R*

This scope comprises the program execution from the start of computation until the occurrence of R. The state/event R is not included in the defined interval, which occurs once in the computation.

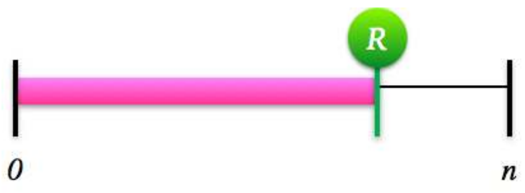


Figure 2.4: *Before* R *Scope*

Before *R* scope characteristics [SGRE11]

1. The scope denotes a sub-sequence of states or events (an interval) that begins with the start of computation and ends with the state or event immediately preceding the event or state at which R holds for first time in the computation;
2. The interval does not include the state or event associated with R;
3. The interval defined by the scope occurs once in a computation;
4. One or more events (conditions) may be associated with R; a condition is a proposition and an event is a change in value of the proposition from one state to the next.

After *L*

This scope comprises the program execution that occurs after L holds. The state/event L is included in the defined interval, which occurs once in the computation.

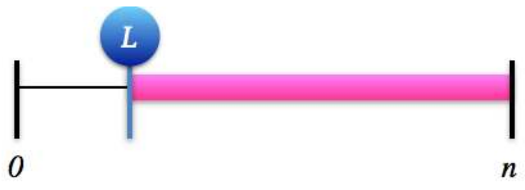


Figure 2.5: *After L Scope*

After *L* scope characteristics [SGRE11]

1. The scope denotes a sub-sequence of states or events (an interval) that begins with the first event or state at which L holds and ends with termination of computation;
2. The interval includes the state or event associated with L;
3. The interval defined by the scope occurs once in a computation;
4. One or more events (conditions) may be associated with L; a condition is a proposition and an event is a change in value of the proposition from one state to the next.

Between *L* and *R*

This scope comprises any part of the execution from L to R. The interval defined by this scope includes the state/event L but it does not include the state/event R. And if L occurs more than once before R, multiple intervals may be defined during one program execution.

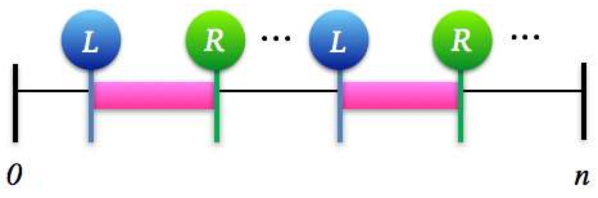


Figure 2.6: *Between L and R Scope*

Between *L* and *R* scope characteristics [SGRE11]

1. The scope denotes a sub-sequence of states or events (an interval) that begins when L holds and ends with the state or event immediately preceding the event or state at which R holds;
2. Event or condition L must hold and, at a different event or state in the future, R must hold;
3. The interval includes the state or event associated with L;
4. The interval does not include the state or event associated with R;
5. The interval defined by the scope may occur more than once in a computation;
6. Multiple intervals may be defined within an interval when L holds more than once before R holds;
7. One or more events (conditions) may be associated with L and R.

After *L* until *R*

This scope is similar to the scope *between* L *and* R. The difference is that the execution, which started after L have occurred, continues even if R does not occur.

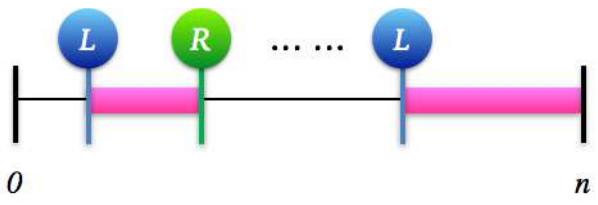


Figure 2.7: *After L until R Scope*

After *L* until *R* scope characteristics [SGRE11]

1. The scope denotes a sub-sequence of states or events (an interval) that begins when L holds and ends either with the state or event immediately preceding the event or state at which R holds, or begins when L holds and ends with the termination of computation;
2. The interval includes the state or event associated with L;

3. The interval does not include the state or event associated with R;
4. The interval may repeat during a computation;
5. If L holds and R does not hold, the interval ends with termination of a computation;
6. The interval defined by the scope may occur more than once in a computation;
7. Multiple intervals may be defined within an interval when L holds more than once before R holds;
8. One or more events (conditions) may be associated with L and R.

2.4.2 Patterns

Patterns are parameterizable specification abstractions which can be mapped to various formalisms, including LTL and regular expressions. In their work, Dwyer et al. organized the patterns in a hierarchy based on their semantics (Figure 2.8). Although there are eight patterns in total, we used in our research five patterns: absence, universality, existence, precedence and response. We chose to work with these five patterns due to their simplicity and statistical evidence that they are the most used (Section 2.4.5). A brief description of the five patterns is presented next [DAC99, Pat, SGRE11].

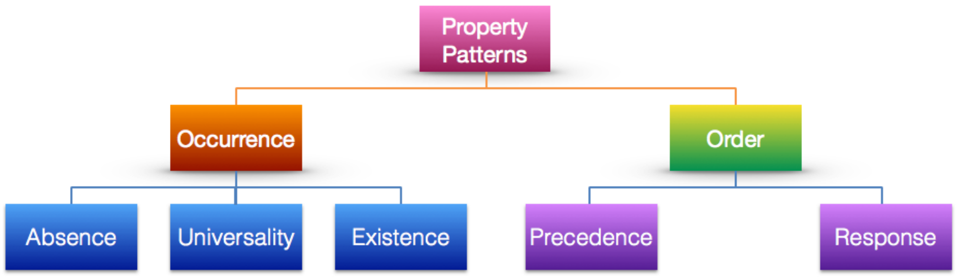


Figure 2.8: *Pattern hierarchy, adapted from [DAC99, Pat]*

Occurrence Patterns

Absence. It states that a given state/event P does not occur within a scope.

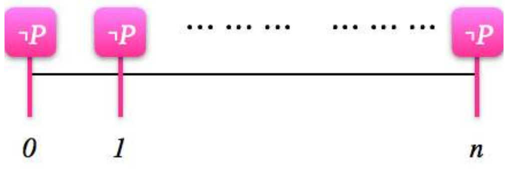


Figure 2.9: *Absence pattern - Global scope*

Universality. It states that a given state/event P occurs throughout a scope.

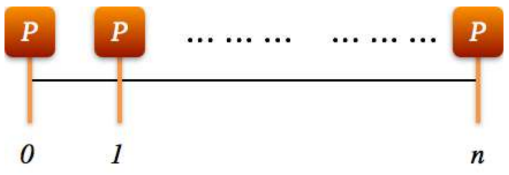


Figure 2.10: *Universality pattern - Global scope*

Existence. It states that a given state/event P must occur (at least once) within a scope.

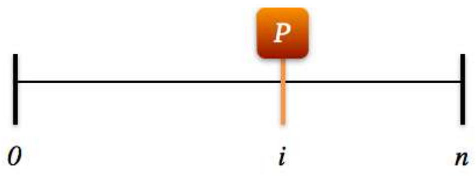
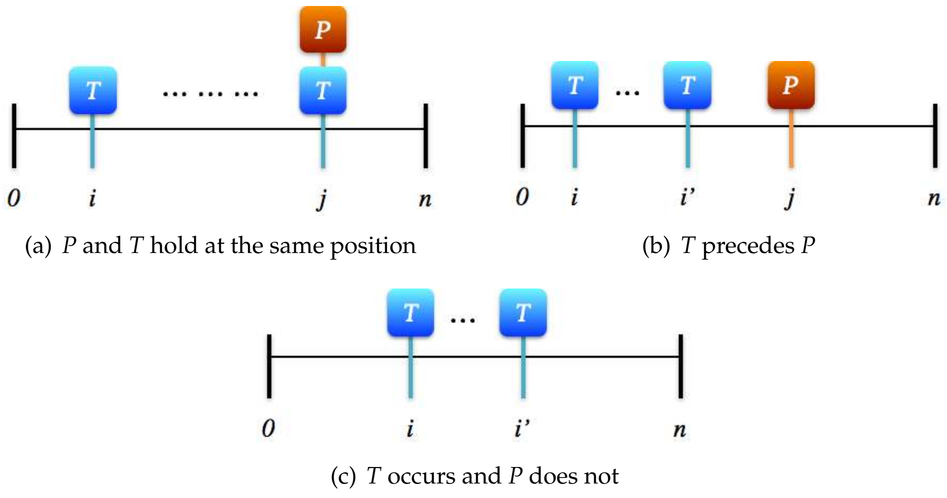


Figure 2.11: *Existence pattern - Global scope*

Order Patterns

Precedence. A state/event *P* must always be preceded by a state/event *T* within a scope. P cannot hold before T holds, and T may occur many times before the occurrence of P. Moreover, if T holds, P may not occur in the computation.

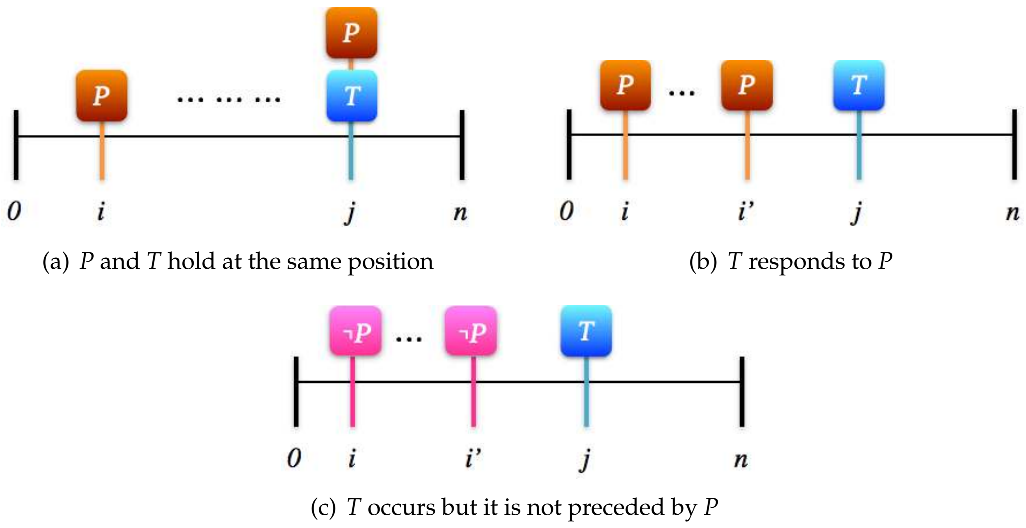


(a) *P* and *T* hold at the same position (b) *T* precedes *P*

(c) *T* occurs and *P* does not

Figure 2.12: *Order Pattern: Precedence - Global Scope*

Response. A state/event *P* must always be followed by a state/event *T* within a scope. P may occur many times before the occurrence of T and, if T holds, P may or may not occur at a previous state in the computation.



(a) *P* and *T* hold at the same position (b) *T* responds to *P*

(c) *T* occurs but it is not preceded by *P*

Figure 2.13: *Order Pattern: Response - Global Scope*

2.4.3 SPS LTL formulas and Regular Expressions

The combinations of SPS patterns and scopes can be mapped to several specification languages and formal representations. Thus, to specify a property using the SPS, practitioners can choose the most suitable combination of pattern and scope in one of the available formal representations (the complete set of SPS mappings to various formalisms can be viewed in [Pat]). In our work, we used two representations of SPS: the SPS LTL formulas (Table 2.3) and the SPS regular expressions (Table 2.4). The LTL formulas were used in our study of properties testability (Part II), while the regular expressions were used with model checking (Part III). Although there is no direct conversion between LTL and regular expressions, they were both used to represent the same expected behavior. The validation of SPS LTL formulas and regular expressions was done through empirical evaluation.

Next, we present in Table 2.3 the LTL formulas for the patterns absence, existence, universality, precedence and response in all SPS scopes. For LTL basic concepts, please refer to Section 2.3.2.

Table 2.3: *SPS - LTL formulas for Patterns and Scopes [Pat]*

Pattern	**Scope**	**LTL Formula**
Absence (P)	Global	$\Box(\neg P)$
	Before R	$\Diamond R \rightarrow (\neg P \ \mathcal{U}\ R)$
	After L	$\Box(L \rightarrow \Box(\neg P))$
	Between L and R	$\Box((L \wedge \neg R \wedge \Diamond R) \rightarrow (\neg P \ \mathcal{U}\ R))$
	After L until R	$\Box(L \wedge \neg R \rightarrow (\neg P \mathcal{W} R))$
Existence (P)	Global	$\Diamond(P)$
	Before R	$\neg R \mathcal{W}(P \wedge \neg R)$
	After L	$\Box(\neg L \vee \Diamond(L \wedge \Diamond P))$
	Between L and R	$\Box(L \wedge \neg R \rightarrow (\neg R \mathcal{W}(P \wedge \neg R)))$
	After L until R	$\Box(L \wedge \neg R \rightarrow (\neg R \ \mathcal{U}\ (P \wedge \neg R)))$
Universality (P)	Global	$\Box(P)$
	Before R	$\Diamond R \rightarrow (P \ \mathcal{U}\ R)$
	After L	$\Box(L \rightarrow \Box(P))$
	Between L and R	$\Box((L \wedge \neg R \wedge \Diamond R) \rightarrow (P \ \mathcal{U}\ R))$
	After L until R	$\Box(L \wedge \neg R \rightarrow (P \mathcal{W} R))$
Precedence (T, P) (T Precedes P)	Global	$\neg P \mathcal{W} T$
	Before R	$\Diamond R \rightarrow (\neg P \ \mathcal{U}\ (T \vee R))$
	After L	$\Box \neg L \vee \Diamond(L \wedge (\neg P \mathcal{W} T))$
	Between L and R	$\Box((L \wedge \neg R \wedge \Diamond R) \rightarrow (\neg P \ \mathcal{U}\ (T \vee R)))$
	After L until R	$\Box(L \wedge \neg R \rightarrow (\neg P \mathcal{W}(T \vee R)))$
Response (T, P) (T Responds to P)	Global	$\Box(P \rightarrow \Diamond T)$
	Before R	$\Diamond R \rightarrow (P \rightarrow (\neg R \ \mathcal{U}\ (T \wedge \neg R))) \ \mathcal{U}\ R$
	After L	$\Box(L \rightarrow \Box(P \rightarrow \Diamond T))$
	Between L and R	$\Box((L \wedge \neg R \wedge \Diamond R) \rightarrow$ $(P \rightarrow (\neg R \ \mathcal{U}\ (T \wedge \neg R))) \ \mathcal{U}\ R)$
	After L until R	$\Box(L \wedge \neg R \rightarrow ((P \rightarrow (\neg R \ \mathcal{U}\ (T \wedge \neg R)))\mathcal{W} R)$

Below, we present Table 2.4, which containg the regular expressions for the patterns absence, universality, existence, precedence and response in all SPS scopes. Note that using regular expressions, we deal only with events in a finite execution context. However, regular expressions can be easily converted into automata representation, which is very useful for model checking.

Table 2.4: *SPS - Regular Expressions for Patterns and Scopes [Pat]*

Pattern	Scope	LTL Formula
Absence (P)	Global	`[−P]*`
	Before R	`[−R]* | [−P, R]*; R; .*`
	After L	`[−L]*; (L; [−P]*)?`
	Between L and R	`([−L]*; L; [−P, R]*; R)*; [−L]*; (L; [−R]*)?`
	After L until R	`([−L]*; L; [−P, R]*; R)*; [−L]*; (L; [−P, R]*)?`
Universality (P)	Global	`[−N]*`
	Before R	`[−R]* | [−N, R]*; R; .*`
	After L	`[−L]* | ([−L]*; P)?; [−N, L]*; L; [−N]*`
	Between L and R	`(([−L]*; P)?; [−N, L]*; L; [−N, R]*; R)*; [−L]*; (L; [−R]*)?`
	After L until R	`(([−L]*; P)?; [−N, L]*; L; [−N, R]*; R)*; ([−L]* | ([−L]*; P)?; [−N, L]*; L; [−N, R]*)`
Precedence (T, P) (T Precedes P)	Global	`[−P]* | ([−T, P]*; T; .*)`
	Before R	`[−R]* | ([−P, R]*; R; .*) | ([−T, P, R]*; T; .*)`
	After L	`[−L]*; (L; ([−P]* | ([−T, P]*; T; .*)))?`
	Between L and R	`[−L]*; (L; [−P, R]* | ([−T, P, R]*; T; [−R]*)R; [−L]*)*; (L; [−R]*)?`
	After L until R	`[−L]*; (L; [−P, R]* | ([−T, P, R]*; T; [−R]*)R; [−L]*)*; (L; ([−P, R]* | ([−T, P, R]*; T; [−R]*)))?`
Existence (P)	Global	`[−P]*; P; .*`
	Before R	`[−R]* | [−P, R]*; P; .*`
	After L	`[−L]*; (L; [−P]*; P; .*)?`
	Between L and R	`([−L]*; L; [−P, R]*; P; [−R]*; R)*; [−L]*; (L; [−R]*)?`
	After L until R	`([−L]*; L; [−P, R]*; P; [−R]*; R)*; [−L]*; (L; [−P, R]*; P; [−R]*)?`
Response (T, P) (T Responds to P)	Global	`[−P]*; (P; [−T]*; T; [−P]*)*`
	Before R	`[−R]* | [−P, R]*; (P; [−T, R]*; T; [−P, R]*)*; R; .*`
	After L	`[−L]*; (L; [−P]*; (P; [−T]*; T; [−P]*)*)?`
	Between L and R	`[−L]*; (L; [−P, R]*; (P; [−T, R]*; T; [−P, R]*)*; R; [−L]*)*; (L; [−R]*)?`
	After L until R	`[−L]*; (L; [−P, R]*; (P; [−T, R]*; T; [−P, R]*)*; R; [−L]*)*; (L; [−P, R]*; (P; [−T, R]*; T; [−P, R]*)*)?`

Remark about Regular Expressions for the Universality Pattern. As we could observe in Table 2.4, the regular expressions for the universality pattern use the letter N. This letter is used to represent the negative event. As stated in [Pat]:

"Specifying universality in an event-based formalism requires the identification of complementary events. The positive event, P*, is defined such that after an occurrence of* P *states have the property. The negative event,* N*, is defined such that after an occurrence of* N *states fail to have the property. Then we define universality throughout a scope as having seen a* P *prior to the beginning of the scope with no* N *until after the close of the scope."*

2.4.4 Specifying Properties using SPS

Previously, we have shown how to specify properties using LTL and regular expressions (Sections 2.3.2 and 2.3.3). In this section, we show how the SPS simplifies this process: now, even if a developer is not an expert in LTL and regular expressions, he or she can specify properties in these formalisms using the formulas that are provided by SPS. The only requirement is to identify the SPS pattern and scope that are most suitable for the property.

Recalling the program `ArithmeticExample` (Section 2.3.1), we specify the same properties that were presented in Sections 2.3.2 and 2.3.3, and specify one additional property using SPS LTL formulas and regular expressions. These properties concern to the program events `add1`, `reset` and `isZero`.

■ **Example 2.4.1** Property: *"the method* `add1` *must be called at least once during the program execution"*. This property states that there is at least one call to the method `add1` for every program execution trace. This property can be specified in LTL using the existence pattern (global scope) of Table 2.3: $\Diamond$ P. Replacing P by `add1` in the formula, we have the specification of the property:

$$\Diamond \, \texttt{add1}$$

■ **Example 2.4.2** Property: *"when the method* `reset` *is called, the method* `isZero` *is called in the future, i.e., a* `reset` *method call is followed by a call to* `isZero`*"*. To specify this property as a regular expression, we use the response pattern (global scope) of Table 2.4:

$$[-\texttt{P}]^*;\ (\texttt{P};\ [-\texttt{T}]^*;\ \texttt{T};\ [-\texttt{P}]^*)^*$$

Replacing P by reset and T by isZero, we get the property specification:

$$[-\texttt{reset}]^*;\ (\texttt{reset};\ [-\texttt{isZero}]^*;\ \texttt{isZero};\ [-\texttt{reset}]^*)^*$$

■ **Example 2.4.3** Property: *the value of* x *is checked at least once after the* reset *method is called.* In this property, we have a scope that is limited by the occurrence of `reset`. Within this scope, there is at least one call to `isZero` to check the value of *x*. Therefore, to specify this property, we have the *existence* pattern for `P = isZero`; and, the *after L* scope for `L = reset`. Using the formula for the existence pattern (after L scope) from Table 2.3, we get the specification of the property in LTL by replacing P for `isZero` and L for `reset`:

$$\Box(\neg(\text{reset}) \vee \Diamond(\text{reset} \wedge \Diamond\text{isZero}))$$

And using the same approach for the existence pattern (after L scope) formula from Table 2.4, we can represent the same property using a regular expression:

$$[-\texttt{reset}]^*;\ (\texttt{reset};\ [-\texttt{isZero}]^*;\ \texttt{isZero};\ .^*)?$$

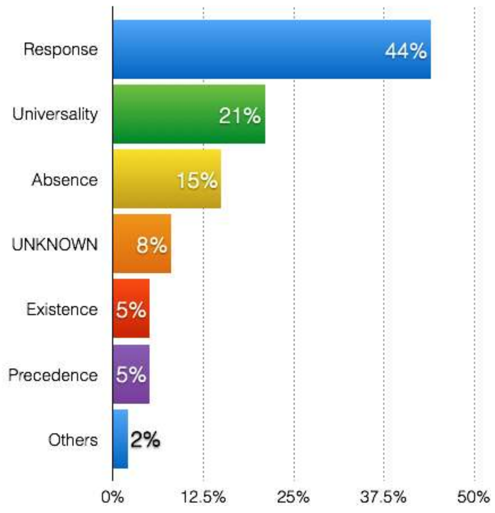
2.4.5 Statistical Data and Evidences

To demonstrate that with the SPS software developers could express most of the system properties specification formally, even if they are not experts in formal methods and techniques, the SPS authors carried out a survey in which 555 specification examples were analyzed. Table 2.5 presents the data collected.

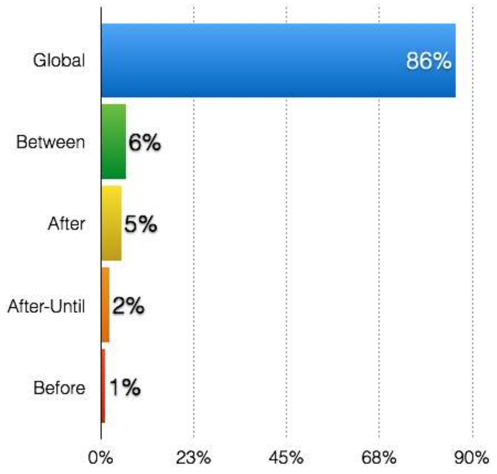
Table 2.5: *Totals for Patterns/Scopes, as presented in [DAC99].*

	Scopes					
Patterns	Global	Before	After	Between	After-Until	**Total**
Absence	41	5	12	18	9	85
Universality	110	1	5	2	1	119
Existence	12	1	4	8	1	26
Bound Exist	0	0	0	1	0	1
Response	241	1	3	0	0	245
Precedence	25	0	1	0	0	26
Resp. Chain	8	0	0	0	0	8
Prec. Chain	1	0	0	0	0	1
UNKNOWN						44
Total	438	8	25	29	11	555

The empirical evidences that were found – 92% (or 511/555) of the examples analyzed matched one of the SPS patterns – confirmed the fact that the majority of properties specification could be expressed as instances of the patterns and scopes described in the specification system proposed by Dwyer et al. Furthermore, the data presented that most of the specification examples used the global scope and the patterns response, universality, absence, existence and precedence, which correspond to the patterns that we chose to work with in our research. Figure 2.14 shows the percentage of scope and pattern occurrences in the specification examples that were analyzed on the survey.



(a) Percentage of pattern occurrences



(b) Percentage of scope occurrences

Figure 2.14: *Graphs with the percentage of scope and pattern occurrences according to the data presented in [DAC99]*

2.5 Conclusion

This research is inserted in the context of programs *formal verification* (Section 2.1). The main objective of the work is to assist practitioners in the process of specifying a property in a formal specification language, and then, help in the verification of the specified properties against the system implementation to certify whether the property was satisfied by the program behavior. Particularly, we are interested in working with model checkers, which can provide valuable information even when the property is not satisfied by the system implementation (Section 2.1.1).

As we could observe, it is essential to have good properties to check if the program behaves according to the specification. But there are infinite possibilities to write system properties concerning to its specification. Therefore, we must determine our "*properties universe*" to have a starting point to work with properties specification. After reviewing related work regarding this subject, we realized that the concept of *test purpose* (Section 2.2) can be used for this objective, hence, *every property we consider in our research is a test purpose* (Section 2.2.1).

Having defined our properties universe, the next step was to handle the process of property specification. For this purpose, we described LTL and regular expressions, presenting the basic concepts of each specification language. We use both of them to specify system properties (Section 2.3). *LTL* (Section 2.3.2) has been widely used to design sequential and concurrent systems, and to specify system properties in model-checking algorithms [BK08, MP90]. Since we work in the context of programs with finite execution, the expressiveness of the LTL fit the requirements of our research, although we have more expressive specification languages (e.g. Computational Tree Logic (CTL). *Due to its large use in model checking theory and program specification, the LTL was suitable for our theoretical research, which deals with linear temporal properties classification and properties testability.* This topic is more explored in Part II. On the other hand, *regular expressions* (Section 2.3.3) are more associated with events traces. Using sets of strings, we can represent execution traces we want to observe in a formal verification. In our case, regular expressions are especially useful to represent system properties in practice, since we can use them to check whether a program satisfies the specified property execution trace. Additionally, a regular expression can be easily converted to a minimal DFA, which can be applied in model checking to monitor the verification process of a property. *The use of regular expressions in our research is further discussed in Part III.*

Since we make use of both LTL and regular expressions to represent system properties in a formal manner, it was crucial to have a systematic way to specify properties using these formalisms. The Specification Pattern System (SPS), which was proposed by Dwyer et al. [DAC99], can be used for this purpose (Section 2.4). It reduces the complexity of the properties specification process by defining scopes and general patterns that can be instantiated to specify a system property. Even though we found similar approaches for property specification (Section 1.3), we decided to keep our research focused on the SPS since it can be mapped to both LTL and regular expressions, and it can be applied to simplify properties specifications in the domain of Java programs with finite execution. The use of LTL and regular expressions in our research was discussed in Section 2.3. The domain of our research is justified due to our study regarding SPS and the testability concept; this study, which aims to establish a connection between the SPS and testable properties to create a test purposes classification, is presented in Part II.

Researchers have also developed tools based on the SPS. The most similar to our work is the*Prospec* [MGR04]. It provides a user-friendly interface to define properties in LTL using the SPS. Since we also use LTL in the specification of properties, we used Prospec as an inspiration for our own tool development. The similarities and differences between the Prospec and our work are further discussed in Chapter 6. Moreover, we refer to Prospec in our testability study (Section 4.7 and Appendix C), due to its use of a new set of LTL formulas to represent SPS patterns and scopes.

II— Properties Testability

3 — Testable Properties Classification

In this chapter, we present the study about the testable properties classification and how they can be used with the SPS (Section 2.4). First, we describe the concept of properties testability. Then, we present the classification of LTL properties and the classes of properties that are actually testable. For those properties classes, we present the SPS patterns that can represent each of them regarding the global scope. Finally, we summarize the achieved results and we briefly discuss the scientific contributions concerning to this theoretical study.

3.1 Properties Testability

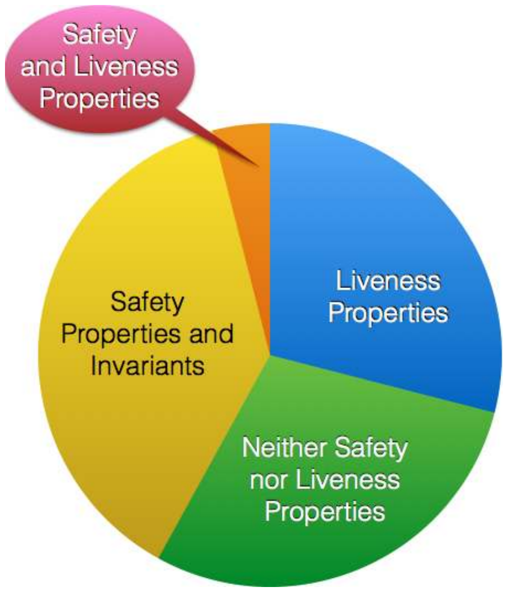
Nahm et al. [NGH93] presented in their work the notion of properties testability by determining the set of linear temporal properties that can be actually tested on a given IUT. According to them[NGH93, FFJ+10], considering a finite test execution σ and a property p that is represented by a temporal formula φ, we can conclude that φ is *testable* if at least one of the four relations between the set of executions satisfying φ and the set of (finite or infinite) executions that could be produced by continuations of σ holds. Naming the set of φ execution sequences as $\mathrm{Tr}(\varphi)$ and the set of IUT execution sequences as $\mathrm{Tr}(\mathrm{IUT})$, we can describe these four relations as [NGH93, FFJ+10]:

- **R1 -** $\mathrm{Tr}(\mathrm{IUT}) \subseteq \mathrm{Tr}(\varphi)$**.** Is the set of execution sequences of the IUT included in the set of execution sequences described by the property?
- **R2 -** $\mathrm{Tr}(\varphi) \subseteq \mathrm{Tr}(\mathrm{IUT})$**.** Is the set of execution sequences described by the property included in the set of execution sequences of the IUT?
- **R3 -** $\mathrm{Tr}(\varphi) = \mathrm{Tr}(\mathrm{IUT})$**.** Is the set of execution sequences of the IUT equal to the set of execution sequences described by the property?
- **R4 -** $\mathrm{Tr}(\varphi) \cap \mathrm{Tr}(\mathrm{IUT}) = \emptyset$**.** Do the set of execution sequences of the IUT and the set of execution sequences described by the property intersect?

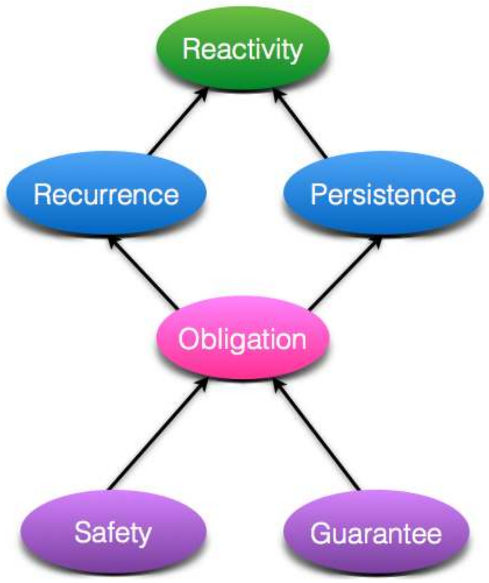
The set of properties that are testable was determined by Nahm et al. using the Safety-Progress classification or the Borel Hierarchy [MP90]. This classification and the set of testable properties are presented in Sections 3.2 and 3.3.1, respectively.

3.2 Linear Temporal Properties Classification

Linear temporal (LT) properties can be categorized according to two main classifications: the *Safety-Liveness (SL) classification* (Figure 3.1(a))[Lam77] and the *Safety-Progress (SP) classification* (Figure 3.1(b)), also known as the *Borel Hierarchy* [MP90]. The first does the partition of properties into three main categories: safety properties, liveness properties, or none of them. The latter, instead of partitioning the properties domain, classifies the properties using a hierarchy composed by the following classes: safety properties, guarantee properties, obligation properties, recurrence properties, persistence properties and reactivity properties.



(a) SL Classification, adapted from [BK08]



(b) SP Classification or the Borel Hierarchy, adapted from [MP90]

Figure 3.1: *LT Properties Classification*

The main difference between the SL classification and the SP classification is the manner used to express non-safety properties. While the first divides them in the partitions of liveness properties or properties that are neither safety nor liveness properties, the latter provides a more refined classification by using a hierarchy of classes. A hierarchical classification does not use partitions; for instance, in the SP classification some non-safety classes contain the safety class. The finer grain classification provided by the Borel Hierarchy allows the use of linear temporal formulas to properly express, besides the safety class, the non-safety classes. Due to this fact, the SP classification was used in the study of the properties testability [NGH93, FFJ$^+$10].

The next sections present the SL classification and the SP classification of LT properties. For basic concepts of LTL, refer to Chapter B.

3.2.1 SL Classification

The Safety-Liveness (SL) classification (Figure 3.1(a)) [Lam77] divides the LT properties domain in four categories: safety properties and invariants, liveness properties, safety and liveness properties, and properties that are neither safety nor liveness properties. Each category is described next[BK08, MP90].

Definition 3.2.1 — Safety Properties and Invariants. A safety property states that some bad thing *never* happens. A particular kind of safety properties is named *invariants* [BK08]. The invariants are LT properties that require that a given condition holds for all reachable states of a program model.

LT Representation. The canonical LT formula that expresses a safety property is: $\Box\varphi$, where φ is a past formula.

Definition 3.2.2 — Liveness Properties. A liveness property states that some good thing *eventually* happens.

LT Representation. The liveness formula is given by: $\Diamond(\bigvee_{i=1}^{n}(\varphi_i \wedge \Diamond\gamma_i))$, where γ_i are satisfiable future formulas and φ_i are past formulas such that $\Box(\bigvee_{i=1}^{n}\varphi_i)$ is a valid formula, $0 < i \leq n$.

Definition 3.2.3 — Safety and Liveness Properties. There is only one kind of property located in the intersection of the safety and liveness properties. It is the nonrestrictive type, which allows all possible behaviors. Properties that belong to this class are the trivial properties of the empty sets and set of all computations.

LT Representation. The nonrestrictive property is the equivalent of $\mathtt{true}$.

Definition 3.2.4 — Properties that are neither Safety nor Liveness Properties. These properties do not fit into the safety partition and into the liveness partition. They cannot be represented by a general LT formula.

An important remark is that safety properties constrain the system finite behaviors and usually represent conditions that should be continuously maintained by the system. On the other hand, the liveness properties constrain the system infinite behaviors and represent conditions whose eventual or repeated occurrence must be guaranteed.

3.2.2 SP Classification

The Safety-Progress (SP) classification or the Borel Hierarchy (Figure 3.1(b)) [MP90] categorizes the LT properties in six categories that are organized in a hierarchical manner: safety properties, guarantee properties, obligation properties, recurrence properties, persistence properties and reactivity properties. Each category will be described using the syntactic LTL view and the automata view. Therefore, we describe some automata concepts first; after that, we describe the SP classes.

Some Automata Concepts

This section presents some concepts used to define the classes of the Borel hierarchy using the automata-view[FFJ+10, MP90].

Streett Automaton. A Streett automaton is a tuple $(Q, q_0, \mathcal{T}, \mathcal{L})$, where:

- Q is a finite set of automaton states;
- q_0 is an initial automaton state;
- $\mathcal{T} = \{t(q_i, q_j) \mid q_i, q_j \in Q\}$ is a set of transition conditions. The transition $t(q_i, q_j)$ states that the automaton may proceed from the state q_i to q_j.
- $\mathcal{L} = ((R_1, P_1), (R_2, P_2), \cdots, (R_k, P_k))$ is a finite list of acceptance sets. For $1 \leq i \leq k$, $R_i \subseteq Q$ and $P_i \subseteq Q$ are the sets of recurrent and persistent states, respectively. Moreover, we use $\bar{R_i}$ and $\bar{P_i}$ to denote the set of non-recurrent and of non-persistent states, respectively.

Plain Automaton. It is a $1-$automaton, where we refer to R_1 and P_1 as R and P, respectively.

Reach(q). It is the set of reachable states from the state q. It can be formally defined as: $\mathsf{Reach}(q) =_{\mathsf{def}} \{q' \in Q \mid \exists\, t(q, q')\}$.

CoReach(X). It is the set of co-reachable states from $X \subseteq Q$. It is formally defined as: $\mathsf{CoReach}(X) =_{\mathsf{def}} \{q \in Q \mid \mathsf{Reach}(q) \cap X \neq \varnothing\}$.

SP Classes

The six categories of LT properties are presented next: safety, guarantee, obligation, recurrence, persistence and reactivity. For each class, an informal description is given, followed by its LT and automata representation [CMP92, FFJ+10, MP90, NGH93].

Definition 3.2.5 — Safety. This property states that some bad thing *never* happens (Section 3.2.1).

LT Representation. The canonical LT formula that expresses a safety property is: $\square\varphi$, where φ is a past formula. This canonical formula states that all positions in a computation satisfies φ. Another interesting example of safety property is represented by the basic precedence formula: $\square(\gamma \rightarrow \blacklozenge\varphi)$,where φ and γ are past formulas. The basic precedence states that if γ ever occurs, then it must be preceded by φ.

Automata Representation (Figure 3.2). The safety property is represented by a plain automaton with no recurrent states ($R = \varnothing$) and no transitions from a non-persistent state to a persistent state ($\bar{P} \nrightarrow P$). Therefore, the run of a safety automaton is accepting if and only if all the states in the run belong are in the set P.

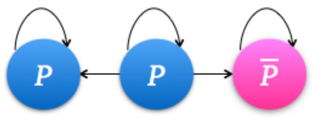


Figure 3.2: *Safety - Streett Automaton Syntactic Restrictions (adapted from [FFJ+10]).*

Definition 3.2.6 — Guarantee. ***Guarantee.*** A guarantee property states that some good thing occurs *at least once*.

LT Representation. The canonical LT formula that expresses a guarantee property is: $\Diamond\varphi$, where φ is a past formula. This canonical formula states that at least one position in a computation satisfies φ.

Automata Representation (Figure 3.3). The guarantee property is represented by a plain automaton with no persistent states ($P = \varnothing$) and no transitions from a recurrent state to a non-recurrent state ($R \nrightarrow \bar{R}$). Thus, this automaton is accepting if and only if finitely many of the states appearing in the run are in the set $\bar{R}$.

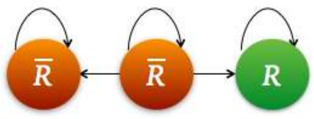


Figure 3.3: *Guarantee - Streett Automaton Syntactic Restrictions (adapted from [FFJ+10]).*

Definition 3.2.7 — Obligation. An obligation property is expressed by a disjunction of safety and guarantee formulas.

LT Representation. A simple obligation formula is: $\Box\varphi \vee \Diamond\gamma$, where φ and γ are past formulas. It states that either φ holds at all positions of a computation or γ holds at some position. The general formula is: $\bigwedge_{i=1}^{n}[\Box\varphi_i \vee \Diamond\gamma_i]$.

Automata Representation. The obligation property is represented by a $k-$automaton such that for $1 \le i \le k$, there is no transition from $\bar{P_i}$ to P_i and from R_i to $\bar{R_i}$ ($\bar{P_i} \nrightarrow P_i, R_i \nrightarrow \bar{R_i}$). The $1-$obligation automaton is denoted as the simple obligation automaton. The definition of the obligation automaton states that once it exits P_i it can never reenter P_i again; and, once it enters R_i it can never get out.

Definition 3.2.8 — Recurrence. A recurrence property states that some good thing occurs infinitely often.

LT Representation. The canonical LT formula that expresses a recurrence property is: $\Box\Diamond\varphi$, where φ is a past formula. This canonical formula states that infinitely many positions in the computation satisfy φ.

Automata Representation (Figure 3.4). The recurrence property is represented by a plain automaton with no persistent states ($P = \varnothing$).

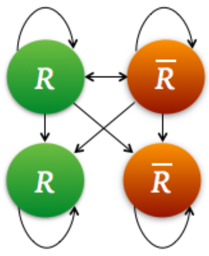


Figure 3.4: *Recurrence - Streett Automaton Syntactic Restrictions (adapted from [FFJ+10]).*

Definition 3.2.9 — Persistence. A persistence property states that some good thing occurs continuously from a certain point on.

LT Representation. The canonical LT formula that expresses a recurrence property is: $\Diamond\Box\varphi$, where φ is a past formula. This canonical formula states that all but finitely many positions in the computation satisfy φ.

Automata Representation (Figure 3.5). The persistence property is represented by a plain automaton with no recurrent states ($R = \varnothing$).

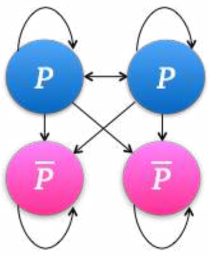


Figure 3.5: *Persistence - Streett Automaton Syntactic Restrictions (adapted from [FFJ+10]).*

Definition 3.2.10 — Reactivity. A reactivity property is expressed by a disjunction of recurrence and persistence formulas.

> *LT Representation.* A simple reactivity formula is: $\Box\Diamond\varphi \vee \Diamond\Box\gamma$, where φ and γ are past formulas. It states that either the computation contains infinitely many φ positions, or all but finitely many of its positions are γ positions. The general formula is: $\bigwedge_{i=1}^{n}[\Box\Diamond\varphi_i \vee \Diamond\Box\gamma_i]$.

> *Automata Representation.* The reactivity property is the most general property. It is represented by an unrestrictive automaton.

In addition, Manna and Pnueli [MP90] discuss these hierarchy classes through other views besides the temporal logic view and the automata view: a language-theoretic view and a topological view. Particularly, in our work we are interested in the temporal logic and automata views since we are using the LTL specification language and the model checking technique.

3.3 Testable Properties

Studying the testability (Section 3.1) of the classes defined by the SP classification [MP90], Nahm et al. [NGH93] reached the conclusion that *the safety and the guarantee properties are testable*. According to them, we can find relations between the execution sequences of a property φ ($\mathrm{Tr}(\varphi)$) and the execution sequences of an IUT ($\mathrm{Tr}(\mathrm{IUT})$) in safety and in guarantee properties. These relations allow us to know, under certain conditions, when a safety property (or a guarantee property) fails.

The results obtained by Nahm et al. were formalized and extended by Falcone et al. [FFJ+10]. In their study, Falcone et al. analyzed the testability relation $\mathrm{Tr}(\mathrm{IUT}) \subseteq \mathrm{Tr}(\varphi)$ (Section 3.1) and concluded that under certain conditions obligation, persistence and recurrence properties are also testable, besides the safety and guarantee properties,.

These SP classification classes can produce a *fail* verdict if the conditions described in Table 3.1 are observed during test execution. These conditions are described in the automata-view of properties (Section 3.2.2)[MP90], and the term $\mathcal{A}_\varphi$ represents the Streett automaton associated with the temporal formula φ that expresses a property.

Table 3.1: *Testable Properties Classes and the Conditions for a Fail Verdict [FFJ+10]*

Property Class	Automata Representation	Testability Condition
Safety	$R = \varnothing, \bar{P} \nrightarrow P$	$\bar{P} \neq \varnothing$
Guarantee	$P = \varnothing, R \nrightarrow \bar{R}$	$\bar{R} \backslash \mathrm{CoReach}_{\mathcal{A}_\varphi}(R) \neq \varnothing$
Obligation	for $1 \leq i \leq k$, $\bar{P}_i \nrightarrow P_i$, $R_i \nrightarrow \bar{R}_i$	$\bigcup_{i=1}^{k} \left(\bar{P}_i \cap \bar{R}_i \backslash \mathrm{CoReach}_{\mathcal{A}_\varphi}(R_i) \neq \varnothing\right)$
Recurrence	$P = \varnothing$	$\bar{R} \backslash \mathrm{CoReach}_{\mathcal{A}_\varphi}(R) \neq \varnothing$
Persistence	$R = \varnothing$	$\bar{P} \backslash \mathrm{CoReach}_{\mathcal{A}_\varphi}(P) \neq \varnothing$

Although they stated the conditions for a *fail* verdict, the conditions for a *pass* verdict were not defined. According to Falcone et al., this occurs because the tester may not know the whole set of $\mathrm{Tr}(\mathrm{IUT})$. However, concerning to a single execution sequence of the IUT, a *weak-pass* verdict can be established. The weak-pass verdict occurs when an execution sequence of the program positively determines the property φ, i.e., it satisfies φ and every finite or infinite continuation of this sequence also satisfy φ. Thus, the weak-pass verdict guarantees that the IUT exhibits behaviors that will satisfy the property φ. Table 3.2 presents the conditions for each property class to produce a weak-pass verdict. These conditions are dual to the conditions for a fail verdict (Table 3.1).

Table 3.2: *Testable Properties Classes and the Conditions for a Weak-Pass Verdict [FFJ+10]*

Property Class	Automata Representation	Testability Condition
Safety	$R = \varnothing, \bar{P} \nrightarrow P$	$P \backslash \mathrm{CoReach}_{\mathcal{A}_\varphi}(\bar{P}) \neq \varnothing$
Guarantee	$P = \varnothing, R \nrightarrow \bar{R}$	$\bar{R} \neq \varnothing$
Obligation	for $1 \leq i \leq k$, $\bar{P_i} \nrightarrow P_i$, $R_i \nrightarrow \bar{R_i}$	$\bigcup_{i=1}^{k} \left(P_i \backslash \mathrm{CoReach}_{\mathcal{A}_\varphi}(\bar{P_i}) \cup R_i \neq \varnothing\right)$
Recurrence	$P = \varnothing$	$R \backslash \mathrm{CoReach}_{\mathcal{A}_\varphi}(\bar{R}) \neq \varnothing$
Persistence	$R = \varnothing$	$P \backslash \mathrm{CoReach}_{\mathcal{A}_\varphi}(\bar{P}) \neq \varnothing$

Analyzing the results obtained by Nahm et al. and Falcone et al., we can conclude that the safety and guarantee properties are testable; and, the obligation, recurrence and persistence properties are also testable under certain conditions.

3.3.1 Testable Properties Representation in SPS (Global Scope)

In our work, we want to relate the testable properties to the SPS (Section 2.4) so that we can define testable properties using abstract patterns. A remark about the researches on testability is that they consider both finite and infinite executions. Since we use the SPS representations in LTL and regular expressions, we decided to focus our research only on programs with finite executions. Therefore, we are considering in this study only the properties that can be tested and/or verified in programs with finite execution. These properties follow the finite trace semantics of LTL (Section B.4).

For a preliminary analysis, we are going to consider the properties satisfied in the whole program execution, i.e., properties that are satisfied under the *SPS global scope*. An example of a property specification within this scope can be found in Section 2.4.4.

The work by Nahm et al. [NGH93] establishes that *safety* and *guarantee* properties are testable. Hence, considering the LT representation of these classes (Section 3.2.2) and the SPS (Table 2.3) we can conclude that the safety properties can be represented by the patterns: universality, absence and precedence; and the guarantee properties can be represented by the existence pattern.

In addition, considering the results of Falcone et. al. [FFJ+10] that some obligation, recurrence and persistence properties are also testable under certain conditions, we have studied how these properties classes can be represented in SPS. The obligation class and the persistence class do not have a direct representation in the SPS, hence they are not discussed here anymore. The recurrence class can be represented by the SPS response pattern. Analyzing the conditions needed to produce verdicts for recurrence properties, we have that:

- A *fail* verdict is produced if the final state of the execution sequence is a non-recurrent state of the Streett automaton associated with the property, and all the possible continuations of this sequence are also non-recurrent states. Since the automaton does not have persistent states and the accepting states are only recurrent states, the recurrence property is not satisfied.

- A *weak-pass* verdict is produced if we can observe that the final state of the execution sequence is a recurrent state of the Streett automaton associated with the property, and all the possible continuations also satisfy the recurrence property.

Considering that in model checking we know (or at least we can establish) the set of execution sequences that are verified against a property, we can check for each execution sequence if it produces a weak-pass verdict or a fail verdict. If one fail verdict is obtained during the verification, we can conclude that the property was violated. In addition, since we are dealing with finite execution programs, we can observe whether the IUT finite execution sequences satisfy (violate) the property within the SPS scope[1]. Thus, combining the fact that we are considering a set of known execution sequences, and that these execution sequences are finite and can be checked within the SPS scopes, we conclude that weak-pass and fail verdicts are sufficient conditions to show that the IUT satisfies or not a recurrence property. For this reason, we conclude that *the recurrence property represented by the SPS response pattern is testable*.

The complete set of testable properties that can be represented using the SPS patterns in the global scope is presented in Table 3.3, where P and T represent the program states/events that can be expressed using temporal formulas. A study about the properties testability in the other SPS scopes is presented in the next section.

Table 3.3: *Testable Properties Classes x SPS (Global Scope)*

Property Class	**SPS Pattern**	**LTL Formula**
Safety	Universality (P)	$\square \mathrm{P}$
	Absence (P)	$\square(\neg \mathrm{P}) \equiv \neg(\Diamond \mathrm{P})$
	Precedence (T,P)	$\square(\mathrm{P} \rightarrow \blacklozenge \mathrm{T}) \equiv \neg \mathrm{P}\, \mathcal{W}\, \mathrm{T}$
Guarantee	Existence(P)	$\Diamond \mathrm{P}$
Recurrence	Response (T,P)	$\square\Diamond(\neg \mathrm{P}\ \mathcal{B}\ \mathrm{T}) \equiv \square(\mathrm{P} \rightarrow \Diamond \mathrm{T})$

3.3.2 Testable Properties Representation in SPS (Other Scopes)

Despite the fact that the global scope is the most used scope when defining a property (Section 2.4.5), the other scopes are useful to describe the system requirements in practice [Pat]. This is because they restrict the IUT execution that should be considered to observe if a test purpose is satisfied or violated by the program. To illustrate the importance of these properties, we present two examples of their use, based on the methods presented in Section 2.3.1. The three methods of this example perform arithmetic operations with the integer variable x: (i) `reset` assigns to the variable x the value 0; (ii) `isZero` checks whether the value of x is zero; and, (iii) `add1`, which adds 1 to the value of x.

■ **Example 3.3.1** Suppose that we want to check the value of x at least once after the `reset` method is called. Therefore, to specify this property, we have the *existence* pattern for `P = isZero`; and, the *after L* scope for `L = reset`. Using the formula from Table 2.3, we get the following LTL property:

$$\square(\neg(\mathrm{reset}) \vee \Diamond(\mathrm{reset} \wedge \Diamond \mathrm{isZero}))$$

■ **Example 3.3.2** Suppose that we want to specify a property that checks if after the `reset` method is called, the `isZero` method is called at least once until the `add1` method is called. In this case, we have the *existence* pattern for `P = isZero`; and, the *after L until R* scope for `L = reset` and `R = add1`. Using the formula from Table 2.3, we get the following statement:

$$\square(\mathrm{reset} \wedge \neg(\mathrm{add1}) \rightarrow (\neg(\mathrm{add1})\ \mathcal{U}\ (\mathrm{isZero} \wedge \neg(\mathrm{add1}))))$$

[1]Remember that the SPS scope restricts the IUT execution that must be observed to decide whether a property is satisfied by the system or not, therefore, we consider only the observable execution sequences within the scope to reach a property verdict.

Testability Analysis of SPS Formulas (Other Scopes)

As we could observe in Table 3.3, the SPS patterns formulas of the testable properties classes in the global scope correspond to their canonical formulas. The formulas for the different scopes do not fit the canonical formulas [DAC99, MP92], as we can observe in Sections 2.4.3 and 3.2.2. Therefore, we conducted a study to verify whether these formulas for different scopes preserve the properties testability of the canonical formulas. In Chapter 4 we present this complete study, including the proof sketches for each formula that allowed us to conclude that the formula keeps the testability of the canonical formula. *The result of this study is that all formulas of SPS patterns (*absence, universality, existence, precedence *and* response*) in the SPS scopes (*before R, after L, between L and R*, and* after L until R*) preserve the testability of their corresponding canonical formulas (*SPS formulas for the global scope*). Therefore, the properties that were presented in the previous examples (Examples 3.3.1 and 3.3.2) are also testable.*

Remark. The conclusion we reached in Sections 3.3.1 and 3.3.2 is that the LTL formulas for SPS patterns (*absence*, *universality*, *existence*, *precedence* and *response*) in the SPS scopes (*global*, *before R*, *after L*, *between L and R*, and *after L until R*) can specify *testable properties*. The proof for the formulas in the global scope is reached directly (Section 3.3.1, Table 3.3); and, for the formulas concerning to the other scopes, the proof is presented in Chapter 4.

3.4 Conclusion

In this chapter, we addressed the problem of finding a classification for test purposes (Section 2.2) based on the concept of testable properties. Initially, we presented the concept of properties testability (Section 3.1), which establishes a connection between program traces and properties verification traces to determine the test purposes that can give a success/fail verdict. After that, a description of the linear temporal properties classification – Safety-Liveness (SL) classification and Safety-Progress (SP) classification – was presented. This latter classification was used to study the classes of testable properties. Finally, by connecting the testability concept to the SPS (Section 2.4), we could establish a set of patterns that can be used to specify testable properties (Table 3.3). In this set, we have the patterns of *universality*, *absence*, *precedence*, *existence* and *response* in their canonical formulas (global scope) and in their formulas for local scopes (*before R*, *after L*, *between L and R*, and *after L until R*). As we could observe in Section 3.3, the connection of SPS patterns in the global scope with the testability concept can be reached directly; however, for the local scopes, a deeper study was required to check whether this result could be extended to those formulas. The study for the local scopes is presented in Chapter 4.

The set of SPS patterns that was achieved in our analysis, which is restricted for programs with finite execution, is our test purposes classification. Although we have this restriction, our proposed test purposes classification simplifies the process of specifying a test purpose, in comparison to works that generates test purposes from a system [dSM06, WFW09]. In addition, it guarantees that the specified properties are testable, thus providing a success/fail verdict when verified against a program with a model checker.

4 — Testable Properties - Local Scopes

According to Table 3.3, the SPS patterns formulas of the testable properties classes in the global scope correspond to their canonical formulas in the Borel Hierarchy (Section 3.2.2). Therefore, the conclusion that the safety class (patterns: *universality*, *absence*, *precedence*), the guarantee class (pattern: *existence*), and the recurrence class (pattern: *response*) are testable in the global scope can be reached directly. However, the formulas for the different scopes do not fit the canonical formulas [DAC99, MP92]. Thus, we need to study if these formulas keep the properties testability in order to guarantee that a property specified using the *universality* pattern in the scope *after L* is still a safety property, for instance. In this study, we analyzed the SPS formulas (Table 2.3) to certify that, besides keeping the properties testability, they have the same semantics.

The study covers the SPS patterns that are able to represent the testable properties classes. For each pattern, we analyzed the LTL formulas in the scopes *before R*, *after L*, *between L and R*, and *after L until R*, using the definitions provided by LTL syntax and semantics (Sections B.2 and B.3). Moreover, we used the following statements in our analysis:

States/Events. The following states/events are used in the analyzed LTL formulas:

- L and R represent the states/events used to limit the scope;
- P and T are used to represent the states/events that should occur in a pattern.

All of these states/events can be represented using temporal formulas.

Remark 1. In our study of properties testability, when we deal with logical implication ($\varphi_1 \rightarrow \varphi_2$, Definition B.3.7) in our analysis (proof sketch) we consider only the situation in which both parts of the implication, φ_1 and φ_2 are *true*. Since we are dealing with scope definitions, if φ_1 is *false* then an occurrence of an state/event like L, R, P or T defined in φ_1 would not happen. But we need them to occur, otherwise, the scope definition would be incomplete. The same occurs to φ_2.

Remark 2. Although R and/or L may occur more than once according to some formulas in scopes *before R* and *after L*, only the first occurrence of the state(s)/event(s) that limit the interval is considered. This restriction apply since the interval defined in these scopes must occur once (Sections 2.4.1 and 2.4.1).

4.1 Safety Properties: Universality Pattern

The universality pattern states that a given state/event occurs throughout a scope. It is a safety property, since in the global scope it corresponds to the canonical formula $\Box P$, where P represents the state/event that must hold. Table 4.1 presents the LTL formulas for this pattern. The ✓ symbol indicates that other works have already shown that the formula preserves the safety property (and therefore it is testable) in the given scope [NGH93, FFJ+10]; and, the ► symbol indicates the formulas that we analyze in the next sections to show that they also preserve the safety property in the given scope.

Table 4.1: *LTL formulas for universality pattern [DAC99]*

Pattern	Scope		SPS
Universality (P)	Global	✓	$\Box(P)$
	Before R	►	$\Diamond R \rightarrow (P\ \mathcal{U}\ R)$
	After L	►	$\Box(L \rightarrow \Box(P))$
	Between L and R	►	$\Box((L \wedge \neg R \wedge \Diamond R) \rightarrow (P\ \mathcal{U}\ R))$
	After L Until R	►	$\Box(L \wedge \neg R \rightarrow (P \mathcal{W} R))$

Before we proceed to the scopes analysis, we give one recurring definition in the proof sketches for this pattern.

Definition 4.1.1 — $P\,\mathcal{U}\,R$. Consider a sequence σ of states $s_0, s_1, \ldots s_n$, where s_0 is the initial state and s_n is the final state of the program execution, and a position j such that $0 \leq j \leq n$. We apply the Definition B.3.19 and get the statement below:

$$(\sigma, j) \vDash P\,\mathcal{U}\,R \quad \text{iff} \quad \exists k,\ j \leq k \leq n,\ \text{s.t. } (\sigma, k) \vDash R \text{ and } \forall i,\ j \leq i < k,\ (\sigma, i) \vDash P$$

This expression states that *P* holds from position *j* until *R* occurs at *k*.

⇒ The Definition 4.1.1 is used in Proof-Sketches 4.1.1, 4.1.3, and 4.1.4.

4.1.1 Before R

Proposition 4.1.1 — Safety Property (Universality Pattern) in the Before R Scope. The *safety* property represented by the *universality* pattern is preserved in the *before R* scope.

$$\Diamond R \rightarrow (P\ \mathcal{U}\ R)$$

Proof Sketch 4.1.1. Consider a sequence σ of states $s_0, s_1, \ldots s_n$, where s_0 is the initial state and s_n is the final state of the program execution. The following statement must hold, according to Definition B.3.7:

$$(\sigma, 0) \vDash \Diamond R \rightarrow (P\ \mathcal{U}\ R) \quad \text{therefore} \quad \text{if } (\sigma, 0) \vDash \Diamond R \text{ then } (\sigma, 0) \vDash (P\ \mathcal{U}\ R)$$

Using this statement, we can analyze each part of the implication as follows:

- **Case 1.** $(\sigma, 0) \vDash \Diamond R \quad \text{iff} \quad (\sigma, k) \vDash R$ for some $k,\ 0 \leq k \leq n$ (Definition B.3.17). This expression deals only with the occurrence of R, which limits the scope. Thus we can assume that R occurs at position k;
- **Case 2.** It is the Definition 4.1.1 for $j = 0$. It states that P holds from the start of the computation until the moment R occurs.

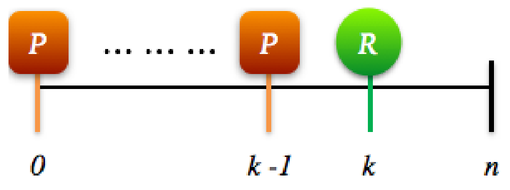


Figure 4.1: *Universality pattern - Before R scope*

Analyzing both parts of the implication, we have that if R occurs at k, $0 \leq k \leq n$, then P occurs $\forall i$, $0 \leq i < k$. Hence, the safety property is preserved in this scope. ∎

4.1.2 After L

Proposition 4.1.2 — Safety Property (Universality Pattern) in the After L Scope. The *safety* property represented by the *universality* pattern is preserved in the *after L* scope.

$$\Box Q, \text{ where } Q = L \rightarrow \Box(P)$$

Proof Sketch 4.1.2. Consider a sequence σ of states $s_0, s_1, \ldots s_n$, where s_0 is the initial state and s_n is the final state of the program execution, and a position j such that $0 \leq j \leq n$. We analyze the Q statement first: $(\sigma, j) \vDash L \rightarrow \Box(P)$.

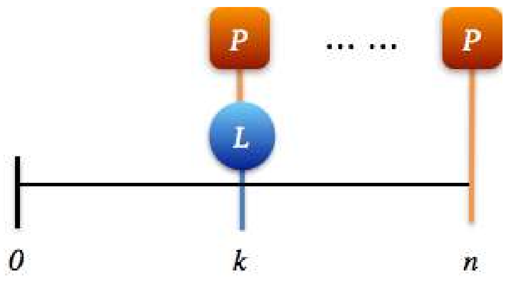


Figure 4.2: *Universality pattern - After L scope*

This statement says that if L occurs at a position k s.t. $j \leq k \leq n$, then P holds $\forall m$ s.t. $k \leq m \leq n$ (Definition B.3.18). Therefore, the safety property is preserved in the Q statement. Now, considering that the original formula is $\Box Q$, we have that every position in which L occurs coincides or is followed by $\Box(P)$. Hence, $\Box Q$ assures that the safety property represented by Q always occurs in the computation.

Remark. Although the SPS original formula does not restrict the occurrence of L, only the first occurrence of the interval is considered since it must occur once (Section 2.4.1).

∎

4.1.3 Between L and R

Proposition 4.1.3 — Safety Property (Universality Pattern) in the Between L and R Scope. The *safety* property represented by the *universality* pattern is preserved in the *between L and R* scope.

$$\Box Q, \text{ where } Q = (L \wedge \neg R \wedge \Diamond R) \rightarrow (P \,\mathcal{U}\, R)$$

Proof Sketch 4.1.3. Consider a sequence σ of states $s_0, s_1, \ldots s_n$, where s_0 is the initial state and s_n is the final state of the program execution, and a position j such that $0 \leq j \leq n$. We first analyze the Q statement, using Definition B.3.7:

$$(\sigma, j) \vDash (L \wedge \neg R \wedge \Diamond R) \rightarrow (P \,\mathcal{U}\, R) \quad \text{therefore} \quad \text{if } (\sigma, j) \vDash L \wedge \neg R \wedge \Diamond R \text{ then } (\sigma, j) \vDash (P \,\mathcal{U}\, R)$$

Using this statement, we can analyze each part of the implication as follows:

- **Case 1.** $(\sigma, j) \models L \wedge \neg R \wedge \Diamond R \quad \text{iff} \quad (\sigma, j) \models L$ and $(\sigma, j) \models \neg R$ and $(\sigma, j) \models \Diamond R$. This expression deals only with the occurrence of L and R, which are the states/events that limit the scope. It says that L occurs at position j and R does not. Moreover, we have that:
  - $(\sigma, j) \models \Diamond R \quad \text{iff} \quad (\sigma, k) \models R$ for some $k > j$ (Definition B.3.17). The position k cannot be equal to position j, otherwise we would have at position j the occurrence of $\neg R \wedge R$, which is a contradiction. Therefore, we can assume that R occurs at position k, $j < k \leq n$.

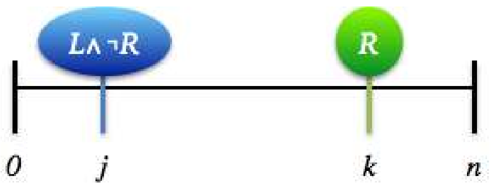


Figure 4.3: *Between L and R scope - Boundaries*

- **Case 2.** Definition 4.1.1.

Analyzing both parts of the implication, we have that if L occurs at j, $0 \leq j \leq n$, R does not occur at j; instead, R occurs at position k, $j < k \leq n$. Then, the model σ satisfies that P occurs $\forall i, j \leq i < k$. Hence, the safety property is preserved in the interval defined by Q statement.

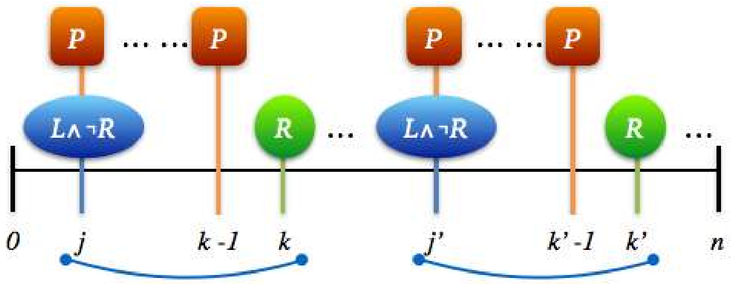


Figure 4.4: *Universality pattern - Between L and R scope - Scope repetition*

Since the universality pattern is represented by $\Box Q$ in this scope, we have that $(\sigma, j) \models \Box Q$ iff $(\sigma, l) \models Q$, $\forall l,\ j \leq l \leq n$ (Definition B.3.18). Hence, $\Box Q$ assures that the safety property represented by Q always occurs in the computation (Figure 4.4).

■

4.1.4 After L Until R

Proposition 4.1.4 — Safety Property (Universality Pattern) in the After L Until R Scope. The *safety* property represented by the *universality* pattern is preserved in the *after L until R* scope.

$$\Box Q, \text{ where } Q = (L \wedge \neg R) \rightarrow (P \mathcal{W} R)$$

Proof Sketch 4.1.4. Consider a sequence σ of states $s_0, s_1, \ldots s_n$, where s_0 is the initial state and s_n is the final state of the program execution, and a position j such that $0 \leq j \leq n$. We first analyze the Q statement, using Definition B.3.7:

$$(\sigma, j) \models L \wedge \neg R \rightarrow (P \mathcal{W} R) \quad \text{therefore} \quad \text{if } (\sigma, j) \models L \wedge \neg R \text{ then } (\sigma, j) \models (P \mathcal{W} R)$$

Using this statement, we can analyze each part of the implication as follows:

- **Case 1.** $(\sigma, j) \vDash L \wedge \neg R$ iff $(\sigma, j) \vDash L$ and $(\sigma, j) \vDash \neg R$. This formula deals with the states/events that limit the scope, and says that L occurs at position j and R does not.
- **Case 2.** $(\sigma, j) \vDash (P \ \mathcal{W} \ R)$ iff $(\sigma, j) \vDash P \mathcal{U} R$ or $\Box P$ (Definition B.3.20). Separating the two cases of this statement:
  - $(\sigma, j) \vDash P \, \mathcal{U} \, R$: this expression states that P holds from position j until R occurs at k (Definition 4.1.1).
  - $(\sigma, j) \vDash \Box P$ iff $(\sigma, k) \vDash P$ for all $j \leq k \leq n$ (Definition B.3.18). This expression states that P holds continuously from position j until the end of computation if R does not occur in the computation.

Analyzing both parts of the implication, we have that if L occurs at j, $0 \leq j \leq n$ and R does not occur at j, then, the model σ satisfies one of these two cases:

- P occurs $\forall i, j \leq i < k$ and R occurs at k s.t. $j \leq k \leq n$.
- R does not occur and P holds $\forall i, j \leq i \leq n$.

Hence, since P holds continuously from the occurrence of L until a future occurrence of R or the end of computation, the safety property is preserved in the interval defined by Q statement.

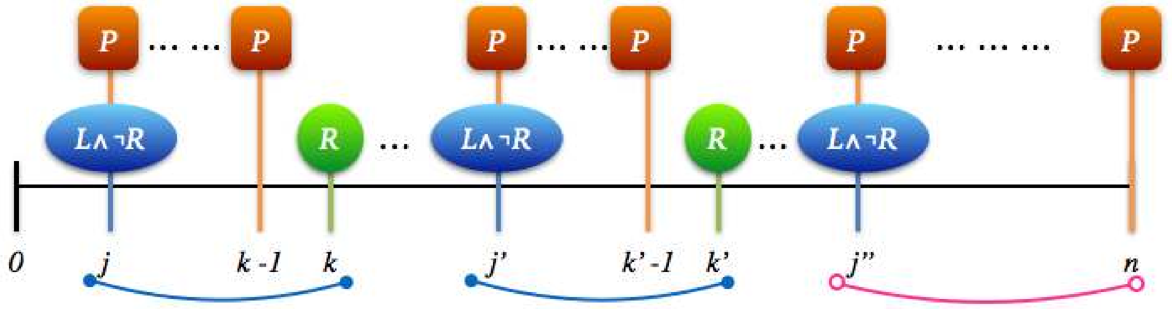


Figure 4.5: *Universality pattern - After L until R scope - Scope repetition*

Since the universality pattern is represented by $\Box Q$ in this scope, we have that $(\sigma, j) \vDash \Box Q$ iff $(\sigma, l) \vDash Q$, $\forall l, \ j \leq l \leq n$ (Definition B.3.18). Hence, $\Box Q$ assures that the safety property represented by Q always occurs in the computation (Figure 4.5). ■

4.2 Safety Properties: Absence Pattern

The absence pattern states that a given state/event does not occur throughout a scope. It is a safety property, since in the global scope it corresponds to the formula $\Box \neg P$, where P represents the state/event that cannot hold. Table 4.2 presents the LTL formulas for this pattern. The ✓ symbol indicates that other works have already shown that the formula preserves the safety property (and therefore it is testable) in the given scope [NGH93, FFJ+10]; and, the ► symbol indicates the formulas that we analyze in the next sections to show that they also preserve the safety property in the given scope.

Remark. The expression $(\sigma, j) \vDash \neg P \ \mathcal{U} \ (R \wedge \neg P)$ states that $\neg P$ holds from position j until R occurs at k, and $\neg P$ also holds at k. In the scope definitions limited by R we consider only the fact that $\neg P$ holds from position j until R occurs at k. The fact that $\neg P$ and R hold at k does not change the scopes semantics significantly since the states/events that take place at the same position of R are not included in the intervals of the scopes [SGRE11] (Section 2.4.1).

Table 4.2: *LTL formulas for absence pattern [DAC99]*

Pattern	Scope	SPS	
Absence (P)	Global	✓	$\Box(\neg P)$
	Before R	►	$\Diamond R \rightarrow (\neg P \, \mathcal{U} \, R)$
	After L	►	$\Box(L \rightarrow \Box(\neg P))$
	Between L and R	►	$\Box((L \land \neg R \land \Diamond R) \rightarrow (\neg P \, \mathcal{U} \, R))$
	After L Until R	►	$\Box(L \land \neg R \rightarrow (\neg P \mathcal{W} R))$

4.2.1 Before R

Proposition 4.2.1 — Safety Property (Absence Pattern) in the Before R Scope. The *safety* property represented by the *absence* pattern is preserved in the *before R* scope.

$$\Diamond R \rightarrow (\neg P \, \mathcal{U} \, R)$$

Proof Sketch 4.2.1. Same as the Proof Sketch 4.1.1, replacing P for ¬P in the analyzed formula. ■

4.2.2 After L

Proposition 4.2.2 — Safety Property (Absence Pattern) in the After L Scope. The *safety* property represented by the *absence* pattern is preserved in the *after L* scope.

$$\neg L \, \mathcal{W} \, (L \land \neg(\Diamond P))$$

Proof Sketch 4.2.2. Same as the Proof Sketch 4.1.2, replacing P for ¬P in the analyzed formula. ■

4.2.3 Between L and R

Proposition 4.2.3 — Safety Property (Absence Pattern) in the Between L and R Scope. The *safety* property represented by the *absence* pattern is preserved in the *between L and R* scope.

$$\Box((L \land \neg R \land \Diamond R) \rightarrow (\neg P \, \mathcal{U} \, R))$$

Proof Sketch 4.2.3. Same as the Proof Sketch 4.1.3, replacing P for ¬P in the analyzed formula. ■

4.2.4 After L Until R

Proposition 4.2.4 — Safety Property (Absence Pattern) in the After L Until R Scope. The *safety* property represented by the *absence* pattern is preserved in the *after L until R* scope.

$$\Box(L \land \neg R \rightarrow (\neg P \mathcal{W} R))$$

Proof Sketch 4.2.4. Same as the Proof Sketch 4.1.4, replacing P for ¬P in the analyzed formula. ■

4.3 Safety Properties: Precedence Pattern

The precedence pattern states that a state/event *P* must always be preceded by a state/event *T* within a scope. It is a safety property, since in the global scope it corresponds to the formula $\Box(P \rightarrow \blacklozenge T) \equiv \neg P \; \mathcal{W} \; T$, where T represents the state/event that must precede the state/event P [MP92]. Table 4.3 presents the LTL formulas for this pattern. The ✓ symbol indicates that other works have already shown that the formula preserves the safety property (and therefore it is testable) in the given scope [NGH93, FFJ+10]; and, the ► symbol indicates the formulas that we analyze in the next sections to show that they also preserve the safety property in the given scope.

Table 4.3: *LTL formulas for precedence pattern [DAC99]*

Pattern	Scope	SPS	
Precedence (T, P)	Global	✓	$\neg P \mathcal{W} T$
	Before R	►	$\Diamond R \rightarrow (\neg P\ \mathcal{U}\ (T \vee R))$
	After L	►	$\Box \neg L \vee \Diamond (L \wedge (\neg P \mathcal{W} T))$
	Between L and R	►	$\Box ((L \wedge \neg R \wedge \Diamond R) \rightarrow (\neg P\ \mathcal{U}\ (T \vee R)))$
	After L Until R	►	$\Box (L \wedge \neg R \rightarrow (\neg P \mathcal{W} (T \vee R)))$

Before we proceed to the scopes analysis, we give one recurring definition in the proof sketches for this pattern.

Definition 4.3.1 — $\neg P\ \mathcal{U}\ (T \vee R)$. Consider a sequence σ of states $s_0, s_1, \ldots s_n$, where s_0 is the initial state and s_n is the final state of the program execution, and a position j such that $0 \leq j \leq n$. We apply the Definition B.3.19 and get the statement :

$$(\sigma, j) \vDash \neg P\ \mathcal{U}\ (T \vee R) \quad \text{iff} \quad \exists l,\ j \leq l \leq n,\ \text{s.t.}\ (\sigma, l) \vDash T \vee R \text{ and } \forall i,\ j \leq i < l,\ (\sigma, i) \vDash \neg P$$

This expression states that $\neg P$ holds from position j until the moment T or R occurs.

⇒ *The Definition 4.3.1 is used in Proof-Sketches 4.3.1, 4.3.3 and 4.3.4.*

Remark. According to Manna and Pnueli [MP92], the following formulas represent the *precedence* pattern (safety property):

- $\Box(P \rightarrow \blacklozenge T)$: this formula has the canonical form of a safety property (Section 3.2.2). It states that for each position j, if P holds at j, $0 \leq j \leq n$, there must be an earlier position k, $0 \leq k \leq j$ satisfying T (Definitions B.3.18 and B.3.24). In other words, every position in which P holds coincides or is preceded by a position in which T holds [MP95].
- $\neg P\ \mathcal{W}\ T$: although it does not have the canonical form of a safety property, it keeps the semantics of a safety property. This formula states that either $\neg P$ holds forever or that it holds until an occurrence of T (Definition B.3.20). In other words, it says that the first occurrence of P must coincide or be preceded by a position in which T occurs [MP92].

Therefore, we have that $\Box(P \rightarrow \blacklozenge T) \equiv \neg P\ \mathcal{W}\ T$.

4.3.1 Before R

Proposition 4.3.1 — Safety Property (Precedence Pattern) in the Before R Scope. The *safety* property represented by the *precedence* pattern is preserved in the *before R* scope.

$$\Diamond R \rightarrow (\neg P\ \mathcal{U}\ (T \vee R))$$

Proof Sketch 4.3.1. Consider a sequence σ of states $s_0, s_1, \ldots s_n$, where s_0 is the initial state and s_n is the final state of the program execution. The following statement must hold:

$$(\sigma, 0) \vDash \Diamond R \rightarrow (\neg P\ \mathcal{U}\ (T \vee R)) \quad \text{therefore} \quad \text{if } (\sigma, 0) \vDash \Diamond R \text{ then } (\sigma, 0) \vDash (\neg P\ \mathcal{U}\ (T \vee R))$$

Using this statement, we can analyze each part of the implication as follows:

- **Case 1.** $(\sigma, 0) \models \Diamond R$ iff $(\sigma, k) \models R$ for some k, $0 \leq k \leq n$ (Definition B.3.17). Thus R occurs at k.
- **Case 2.** It is the Definition 4.3.1 for $j = 0$. It states that $\neg P$ holds from the start of the computation until the moment T or R occurs.

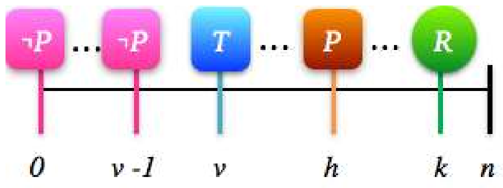


Figure 4.6: *Precedence pattern - Before R scope*

Analyzing both parts of the implication, we can conclude that the following statements must hold:

- $(\sigma, k) \models R$, $0 \leq k \leq n$. It states that R holds at position k and, since T cannot occur at the same time, T occurs some time before R. If T does not occur then P will not occur because we would have that $\neg P$ holds at all positions in the interval $[0, (k-1)]$.
- $(\sigma, j) \models \neg P \, \mathcal{U} \, (T \vee R)$: by Definition B.3.21 we have $(\neg P \, \mathcal{U} \, T) \vee (\neg P \, \mathcal{U} \, R)$ (Definition B.3.5):
  - $(\neg P \, \mathcal{U} \, T)$: T holds at v, $j \leq v < n$, $\neg P$ holds $\forall i, j \leq i < v$, P can hold at some position h, $v \leq h < n$ and R occurs at some position k, $h \leq k \leq n$ (if $v = h = k$, then neither T nor P occurs; if ($v = h < k$), then T and P occur at the same position); or,
  - $(\neg P \, \mathcal{U} \, R)$: R holds at k and $\neg P$ holds for all i, $j \leq i < k$ (i.e. neither T nor P occurs).

Hence, we assure that T always precedes P, and the safety property is preserved in this scope. ■

4.3.2 After L

Proposition 4.3.2 — Safety Property (Precedence Pattern) in the After L Scope. The *safety* property represented by the *precedence* pattern is preserved in the *after L* scope.

$$\Box \neg L \vee \Diamond (L \wedge (\neg P \mathcal{W} T))$$

Proof Sketch 4.3.2. Consider a sequence σ of states $s_0, s_1, \ldots s_n$, where s_0 is the initial state and s_n is the final state of the program execution. We analyze the following statement:

$$(\sigma, 0) \models \Box \neg L \vee \Diamond (L \wedge (\neg P \mathcal{W} T)) \quad \text{iff} \quad (\sigma, 0) \models \Box \neg L \text{ or } (\sigma, 0) \models \Diamond (L \wedge (\neg P \mathcal{W} T)) \quad \text{(Definition B.3.5).}$$

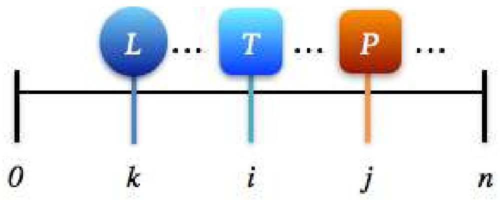


Figure 4.7: *Precedence pattern - After L scope*

Analyzing each part of the disjunction:

- **Case 1.** $(\sigma, 0) \models \Box\neg L$. This formula deals only with the occurrence of L, which limits the scope. It states that $\Box\neg L$ holds for all i, $0 \leq i \leq n$. Thus, L does not occur in the computation, and the scope is not defined.
- **Case 2.** $(\sigma, 0) \models \Diamond(L \wedge (\neg PWT))$ iff $(\sigma, k) \models L \wedge (\neg PWT)$ for some k $0 \leq k \leq n$. Therefore, we have that if $(\sigma, k) \models L$ then $(\sigma, i) \models (\neg PWT)$for some i, $k \leq i \leq n$. This means that when L occurs at k, the precedence pattern $(\neg PWT)$ holds in the interval $[k, n]$.

Considering both cases, we have that the precedence pattern holds only after the occurrence of L. Thus, the safety property is preserved in the scope *after L*.

■

4.3.3 Between L and R

Proposition 4.3.3 — Safety Property (Precedence Pattern) in the Between L and R Scope. The *safety* property represented by the *precedence* pattern is preserved in the *between L and R* scope.

$$\Box Q, \text{ where } Q = (L \wedge \neg R \wedge \Diamond R) \rightarrow (\neg P \ \mathcal{U} \ (T \vee R))$$

Proof Sketch 4.3.3. Consider a sequence σ of states $s_0, s_1, \ldots s_n$, where s_0 is the initial state and s_n is the final state of the program execution, and a position j such that $0 \leq j \leq n$. We first analyze the Q statement:

$$(\sigma, j) \models (L \wedge \neg R \wedge \Diamond R) \rightarrow (\neg P \ \mathcal{U} \ (T \vee R)) \quad \text{therefore} \quad \begin{array}{l} \text{if } (\sigma, j) \models L \wedge \neg R \wedge \Diamond R \\ \text{then } (\sigma, j) \models (\neg P \ \mathcal{U} \ (T \vee R)) \end{array}$$

Using this statement, we can analyze each part of the implication as follows:

- **Case 1.** This expression deals only with the occurrence of L and R, which are the states/events that limit the scope. It says that L occurs at position j and R does not. Moreover, we have that:
  - $(\sigma, j) \models \Diamond R$ iff $(\sigma, k) \models R$ for some $k > j$ (Definition B.3.17). The position k cannot be equal to position j, otherwise we would have at position j the occurrence of $\neg R \wedge R$, which is a contradiction. Therefore, we can assume that R occurs at position k, $j < k \leq n$ (Figure 4.3).
- **Case 2.** Definition 4.3.1

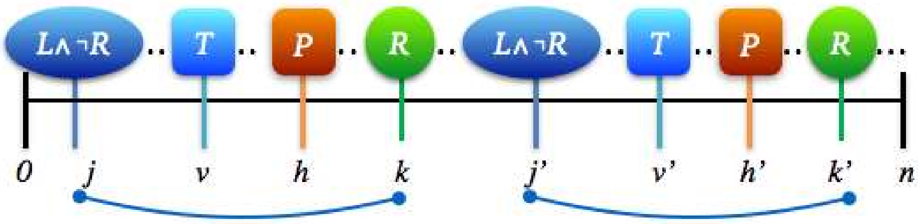


Figure 4.8: *Precedence pattern - Between L and R scope - Scope repetition*

Analyzing both parts of the implication, we have that if L occurs at j, $0 \leq j \leq n$, R does not occur at j; instead, R occurs at position k, $j \leq k \leq n$. And $\neg P$ holds from position j until the moment T or R occurs. This means that:

- $(\sigma, k) \models R$, $j \leq k \leq n$: it states that R holds at position k.
- $(\sigma, j) \models \neg P \ \mathcal{U} \ (T \vee R)$: we analyze $(\neg P \ \mathcal{U} \ T) \vee (\neg P \ \mathcal{U} \ R)$ (Definition B.3.21 (3)).

- $(\neg P \,\mathcal{U}\, T)$: T holds at v, $j \leq v < n$, $\neg P$ holds $\forall i$, $j \leq i < v$, P can hold at some position h, $v \leq h < n$ and R occurs at some position k, $h \leq k \leq n$ (if $v = h = k$, then neither T nor P occurs; if $(v = h < k)$, then T and P occur at the same position); or,
- $(\neg P \,\mathcal{U}\, R)$: R holds at k and $\neg P$ holds for all i, $j \leq i < k$ (i.e. neither T nor P occurs).

- $(\sigma, j) \models L \wedge \neg R$, $0 \leq j \leq n$: it states that L holds at position j and R does not.

Hence, we assure that T always precedes P, and the safety property is preserved in the interval defined by Q statement. Since the precedence pattern is represented by $\Box Q$ in this scope, we have that $(\sigma, j) \models \Box Q$ iff $(\sigma, l) \models Q$,$\forall l$, $j \leq l \leq n$ (Definition B.3.18). Hence, $\Box Q$ assures that the safety property represented by Q always occurs in the computation (Figure 4.8).

■

4.3.4 After L Until R

Proposition 4.3.4 — Safety Property (Precedence Pattern) in the After L Until R Scope. The *safety* property represented by the *precedence* pattern is preserved in the *after L until R* scope.

$$\Box Q, \text{ where } Q = L \wedge \neg R \rightarrow (\neg P \mathcal{W} (T \vee R))$$

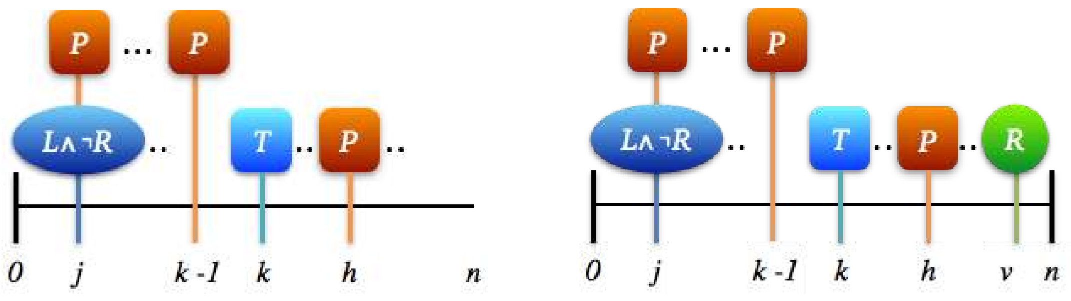
Proof Sketch 4.3.4. Consider a sequence σ of states $s_0, s_1, \ldots s_n$, where s_0 is the initial state and s_n is the final state of the program execution, and a position j such that $0 \leq j \leq n$. We first analyze the Q statement:

$$(\sigma, j) \models L \wedge \neg R \rightarrow (\neg P \;\mathcal{W}\; (T \vee R)) \quad \text{therefore} \quad \text{if } (\sigma, j) \models L \wedge \neg R \text{ then } (\sigma, j) \models (\neg P \;\mathcal{W}\; (T \vee R))$$

Using this statement, we can analyze each part of the implication as follows:

- **Case 1.** $(\sigma, j) \models L \wedge \neg R$ iff $(\sigma, j) \models L$ and $(\sigma, j) \models \neg R$. Thus we can assume that L occurs at position j and R does not.
- **Case 2.** $(\sigma, j) \models (\neg P \;\mathcal{W}\; (T \vee R))$ iff $(\sigma, j) \models (\neg P \;\mathcal{U}\; (T \vee R))$ or $\Box \neg P$. Separating the two cases of this statement:
  - $(\sigma, j) \models \neg P \;\mathcal{U}\; (T \vee R)$ (Definition 4.3.1). This expression states that $\neg P$ holds from position j until T or R occurs at k.
  - $(\sigma, j) \models \Box \neg P$ iff $(\sigma, k) \models \neg P$ for all $j \leq k \leq n$. This expression states that $\neg P$ holds continuously from position j until the end of computation.

Analyzing both parts of the implication, we have that if L occurs at j, $0 \leq j \leq n$ and R does not occur at j, then, the model σ satisfies one of these two cases:



(a) R does not occur and T precedes P (b) R occurs in the future and T precedes P

Figure 4.9: *Precedence pattern - After L Until R scope - Analysis*

- R *occurs in the future.* In this case, the following statements must hold:
  - $(\sigma, j) \models L \wedge \neg R$: it states that L holds at position j and R does not.
  - $(\sigma, j) \models \neg P\ \mathcal{U}\ (T \vee R)$: by Definition B.3.21 we analyze $(\neg P\ \mathcal{U}\ T) \vee (\neg P\ \mathcal{U}\ R)$.
    - $(\neg P\ \mathcal{U}\ T)$: T holds at $k, j \le k < n$, $\neg P$ holds $\forall i, j \le i < k$, P can hold at some position $h, k \le h < n$ and R occurs at some position $v, h \le v \le n$ (if $v = h = k$, then neither T nor P occurs; if ($k = h < v$), then T and P occur at the same position).
    - $(\neg P\ \mathcal{U}\ R)$: R holds at k and $\neg P$ holds for all $i, j \le i < k$ (i.e. neither T nor P occurs).

- R *does not occur in the future.* Then, the following statements must hold:
  - $(\sigma, j) \models L \wedge \neg R$: it states that L holds at position j and R does not.
  - $(\sigma, j) \models \neg P\ \mathcal{U}\ T$: this formula states that T holds at $k, j \le k \le n$, $\neg P$ holds $\forall i, j \le i < k$ and P can hold at some position $h, k \le h \le n$.

Hence, since T precedes P holds continuously from the occurrence of L until a future occurrence of R or the end of computation, the safety property is preserved in the interval defined by Q statement.

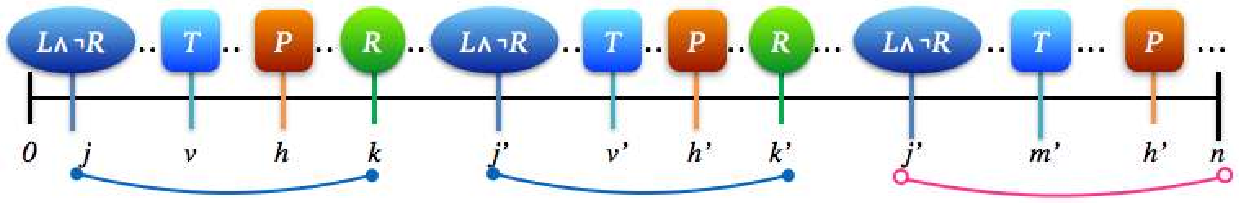


Figure 4.10: *Precedence pattern - After L Until R scope - Scope repetition*

Since the precedence pattern is represented by $\Box Q$ in this scope, we have that $(\sigma, j) \models \Box Q$ iff $(\sigma, l) \models Q$, $\forall l,\ j \le l \le n$ (Definition B.3.18). Hence, $\Box Q$ assures that the safety property represented by Q always occurs in the computation (Figure 4.10).

■

4.4 Guarantee Properties: Existence Pattern

The existence pattern states that a given state/event must occur (at least once) within a scope. It is a guarantee property, since in the global scope it corresponds to the canonical formula $\Diamond P$, where P represents the state/event that must hold. Table 4.4 presents the LTL formulas for this pattern. The ✓ symbol indicates that other works have already shown that the formula preserves the guarantee property (and therefore it is testable) in the given scope [NGH93, FFJ+10]; and, the ► symbol indicates the formulas that we analyze in the next sections to show that they also preserve the guarantee property in the given scope.

Table 4.4: *LTL formulas for existence pattern [DAC99]*

Pattern	Scope		SPS
Existence (P)	Global	✓	$\Diamond(P)$
	Before R	►	$\neg R\,\mathcal{W}(P \wedge \neg R)$
	After L	►	$\Box(\neg L \vee \Diamond(L \wedge \Diamond P))$
	Between L and R	►	$\Box(L \wedge \neg R \rightarrow (\neg R\,\mathcal{W}(P \wedge \neg R)))$
	After L Until R	►	$\Box(L \wedge \neg R \rightarrow (\neg R\ \mathcal{U}\ (P \wedge \neg R)))$

Before we proceed to the scopes analysis, we give one recurring definition in the proof sketches for this pattern.

Definition 4.4.1 — ($\neg R\ \mathcal{U}\ (P \wedge \neg R)$). Consider a sequence σ of states $s_0, s_1, \dots s_n$, where s_0 is the initial state and s_n is the final state of the program execution, and a position j such that $0 \leq j \leq n$. We apply the Definition B.3.20 and get the statement below:

$$\begin{array}{lll}(\sigma, j) \vDash (\neg R\ \mathcal{U}\ (P \wedge \neg R)) & \text{iff} & \exists m,\ j \leq m \leq n,\ \text{s.t. } (\sigma, m) \vDash P \wedge \neg R \\ & \text{and} & \forall i,\ j \leq i < m,\ (\sigma, i) \vDash \neg R\end{array}$$

This expression states that $\neg R$ holds from the start of the computation until the moment P occurs, and when P occurs, $\neg R$ occurs at the same position.

⇒ *The Definition 4.4.1 is used in Proof-Sketches 4.4.1, 4.4.3 and 4.4.4.*

4.4.1 Before R

Proposition 4.4.1 — Guarantee Property (Existence Pattern) in the Before R Scope. The *guarantee* property represented by the *existence* pattern is preserved in the *before R* scope.

$$\neg R\ \mathcal{W}\ (P \wedge \neg R) \equiv \Diamond R \rightarrow \neg R\ \mathcal{U}\ (P \wedge \neg R)$$

Proof Sketch 4.4.1. Consider a sequence σ of states $s_0, s_1, \dots s_n$, where s_0 is the initial state and s_n is the final state of the program execution. The following statement must hold, according to Definition B.3.7:

$$(\sigma, 0) \vDash \Diamond R \rightarrow \neg R\ \mathcal{U}\ (P \wedge \neg R) \quad \text{therefore} \quad \text{if } (\sigma, 0) \vDash \Diamond R \text{ then } (\sigma, 0) \vDash \neg R\ \mathcal{U}\ (P \wedge \neg R)$$

Using this statement, we can analyze each part of the implication as follows:

- **Case 1.** $(\sigma, 0) \vDash \Diamond R$ iff $(\sigma, k) \vDash R$ for some k, $0 \leq k \leq n$ (Definition B.3.17). This expression deals only with the occurrence of R, which limits the scope. Thus we can assume that R occurs at position k;
- **Case 2.** Definition 4.4.1, for $j = 0$.

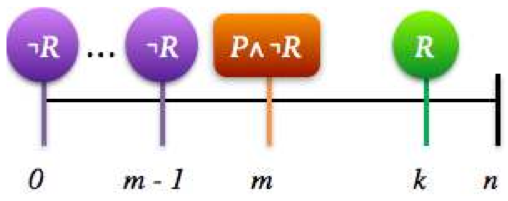


Figure 4.11: *Existence pattern - Before R scope*

The link between the two parts of the implication is that if $P \wedge \neg R$ occurs at m, $0 \leq m < n$ then $\neg R$ holds for all i, $0 \leq i < m$. Since $\neg R$ occurs from position 0 to m, R must occur at position k, $m < k \leq n$. In other words, we have three statements regarding this formula that must hold:

- $(\sigma, m) \vDash P \wedge \neg R$, $0 \leq m < n$. At this position, P occurs and R does not. It guarantees that P occurs at least once in this scope.
- $(\sigma, i) \vDash \neg R, \forall i, 0 \leq i < m$.

- $(\sigma, k) \vDash R$, $m < k \leq n$. This states that R occurs at position k. If $k > (m+1)$ then P may or may not occur in the interval delimited by $[(m+1), k[$. The observation of the occurrence of P here does not influence the conclusion that the property expressed by the existence pattern in this scope is a guarantee property. In addition, R cannot occur in the interval $[(m+1), k[$ because its first occurrence takes place at position k.

Hence, we conclude that the guarantee property is preserved in this scope. ∎

4.4.2 After L

Proposition 4.4.2 — Guarantee Property (Existence Pattern) in the After L Scope. The *guarantee* property represented by the *existence* pattern is preserved in the *after L* scope.

$$\Box Q, \text{ where } Q = \neg L \vee \Diamond(L \wedge \Diamond P)$$

Proof Sketch 4.4.2. Consider a sequence σ of states $s_0, s_1, \dots s_n$, where s_0 is the initial state and s_n is the final state of the program execution, and a position j such that $0 \leq j \leq n$. We analyze the Q statement first: $(\sigma, j) \vDash \neg L \vee \Diamond(L \wedge \Diamond P)$.

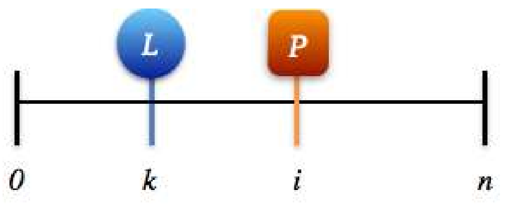


Figure 4.12: *Existence pattern - After L scope*

Analyzing each part of the disjunction:

- **Case 1.** $(\sigma, j) \vDash \neg L$. This formula deals only with the occurrence of L, which limits the scope. It states that $\neg L$ holds at position j, $0 \leq j \leq n$. Thus, L does not occur in the computation, and the scope is not defined.

- **Case 2.** $(\sigma, j) \vDash \Diamond(L \wedge \Diamond P)$ iff $(\sigma, k) \vDash L \wedge \Diamond P$ for some k, $0 \leq k \leq n$. Therefore, we have that if $(\sigma, k) \vDash L$ then $(\sigma, k) \vDash \Diamond P$ iff $(\sigma, i) \vDash P$ for some i, $k \leq i \leq n$

Considering both cases, we have that P occurs at least once after the occurrence of L or at the same position in which L occurs. Thus, the guarantee property is preserved in the Q statement. Since the existence pattern is represented by $\Box Q$ in this scope, we have that $(\sigma, j) \vDash \Box Q$ iff $(\sigma, l) \vDash Q$,$\forall l$, $j \leq l \leq n$ (Definition B.3.18). Hence, $\Box Q$ assures that the guarantee property represented by Q always occurs in the computation (Figure 4.12).

∎

4.4.3 Between L and R

Proposition 4.4.3 — Guarantee Property (Existence Pattern) in the Between L and R Scope. The *guarantee* property represented by the *existence* pattern is preserved in the *between L and R* scope.

$$\Box Q, \text{ where } Q = L \wedge \neg R \rightarrow (\neg R \mathcal{W} (P \wedge \neg R))$$

Proof Sketch 4.4.3. Consider a sequence σ of states $s_0, s_1, \dots s_n$, where s_0 is the initial state and s_n is the final state of the program execution, and a position j such that $0 \leq j \leq n$. We first analyze the Q statement using Definition B.3.7:

$$(\sigma, j) \vDash L \wedge \neg R \rightarrow (\neg R \mathcal{W} (P \wedge \neg R)) \quad \text{therefore} \quad \text{if } (\sigma, j) \vDash L \wedge \neg R \text{ then } (\sigma, j) \vDash (\neg R \; \mathcal{W} \; (P \wedge \neg R))$$

Using this statement, we can analyze each part of the implication as follows:

- **Case 1.** $(\sigma, j) \models L \wedge \neg R$ iff $(\sigma, j) \models L$ and $(\sigma, j) \models \neg R$. This expression deals only with the occurrence of L and R, which are the states/events that limit the scope. It says that L occurs at position j and R does not.
- **Case 2.** Proof Sketch 4.4.1 for the *existence* pattern in the *before R* scope.

Analyzing both parts of the implication, we have that the next four statements regarding this formula must hold:

- $(\sigma, j) \models L \wedge \neg R, 0 \leq j < n$. This states that L occurs at position j and R does not.
- $(\sigma, m) \models P \wedge \neg R, j \leq m < n$. At this position, P occurs and R does not. It guarantees that P occurs at least once in this scope.
- $(\sigma, i) \models \neg R, \forall i, j \leq i < m$.
- $(\sigma, k) \models R, m < k \leq n$. This states that R occurs at position k. If $k > (m+1)$ then P may or may not occur in the interval delimited by $[(m+1), k[$. The observation of the occurrence of P here does not influence the conclusion that the property expressed by the existence pattern in this scope is a guarantee property. In addition, R cannot occur in the interval $[(m+1), k[$ because its first occurrence takes place at position k.

Hence, since $\Diamond P$ holds in the interval $[j, (k-1)]$, the guarantee property is preserved in the Q statement.

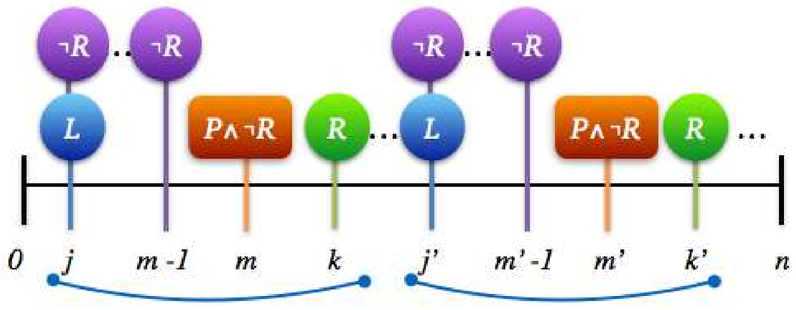


Figure 4.13: *Existence pattern - Between L and R scope - Scope repetition*

Since the existence pattern is represented by $\Box Q$ in this scope, we have that $(\sigma, j) \models \Box Q$iff $(\sigma, l) \models Q$ $, \forall l,\ j \leq l \leq n$ (Definition B.3.18). Hence, $\Box Q$ assures that the guarantee property represented by Q always occurs in the computation (Figure 4.13).

■

4.4.4 After L Until R

Proposition 4.4.4 — Guarantee Property (Existence Pattern) in the After L Until R Scope. The *guarantee* property represented by the *existence* pattern is preserved in the *after L until R* scope.

$$\Box Q, \text{ where } Q = L \wedge \neg R \rightarrow (\neg R\ \mathcal{U}\ (P \wedge \neg R))$$

Proof Sketch 4.4.4. Consider a sequence σ of states $s_0, s_1, \ldots s_n$, where s_0 is the initial state and s_n is the final state of the program execution, and a position j such that $0 \leq j \leq n$. We first analyze the Q statement using Definition B.3.7:

$$(\sigma, j) \models L \wedge \neg R \rightarrow (\neg R\ \mathcal{U}\ (P \wedge \neg R)) \quad \text{therefore} \quad \text{if } (\sigma, j) \models L \wedge \neg R \text{ then } (\sigma, j) \models (\neg R\ \mathcal{U}\ (P \wedge \neg R))$$

Using this statement, we can analyze each part of the implication as follows:

- **Case 1.** $(\sigma, j) \vDash L \wedge \neg R \quad \text{iff} \quad (\sigma, j) \vDash L$ and $(\sigma, j) \vDash \neg R$. Thus we can assume that L occurs at position j and R does not.
- **Case 2.** Definition 4.4.1.

Analyzing both parts of the implication, we have that the following statements must hold in σ:

- If R occurs at least once after the occurrence of L:
  - $(\sigma, j) \vDash L \wedge \neg R, 0 \leq j < n$. This states that L occurs at position j and R does not.
  - $(\sigma, m) \vDash P \wedge \neg R, j \leq m < n$. At this position, P occurs and R does not. It guarantees that P occurs at least once in this scope.
  - $(\sigma, i) \vDash \neg R, \forall i, j \leq i < m$.
  - $(\sigma, k) \vDash R, m < k \leq n$. This states that R occurs at position k. If $k > (m+1)$ then P may or may not occur in the interval delimited by $[(m+1), k[$. The observation of the occurrence of P here does not influence the conclusion that the property expressed by the existence pattern in this scope is a guarantee property. In addition, R cannot occur in the interval $[(m+1), k[$ because its first occurrence takes place at position k.
- If R does not occur anymore after the occurrence of L:
  - $(\sigma, j) \vDash L \wedge \neg R, 0 \leq j \leq n$. This states that L occurs at position j and R does not.
  - $(\sigma, m) \vDash P \wedge \neg R, j \leq m \leq n$. At this position, P occurs and R does not. It guarantees that P occurs at least once in this scope.
  - $(\sigma, i) \vDash \neg R, \forall i, j \leq i < m$.

Hence, since P holds eventually from the occurrence of L until a future occurrence of R or the end of computation, the guarantee property is preserved in the interval defined by Q statement.

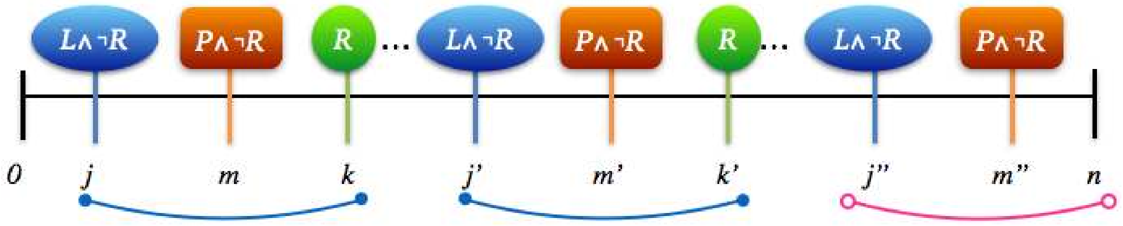


Figure 4.14: *Existence pattern - After L Until R scope - Scope repetition*

Since the existence pattern is represented by $\Box Q$ in this scope, we have that $(\sigma, j) \vDash \Box Q$ iff $(\sigma, l) \vDash Q\ , \forall l,\ j \leq l \leq n$ (Definition B.3.18). Hence, $\Box Q$ assures that the guarantee property represented by Q always occurs in the computation (Figure 4.14).

■

4.5 Recurrence Properties: Response Pattern

The response pattern states that a state/event *P* must always be followed by a state/event *T* within a scope. It is a recurrence property, since in the global scope it corresponds to the formula $\Box\Diamond(\neg P\ \mathcal{B}\ T) \equiv \Box(P \to \Diamond T)$, where T represents the state/event that must respond to the state/event P. Table 4.5 presents the LTL formulas for this pattern. The ✓ symbol indicates that other works have already shown that the formula preserves the recurrence property (and therefore it is testable) in the given scope [NGH93, FFJ+10]; and, the ► symbol indicates the formulas that we analyze in the next sections to show that they also preserve the recurrence property in the given scope.

Table 4.5: *LTL formulas for response pattern [DAC99]*

Pattern	Scope	SPS
Response (T, P)	Global	✓ $\Box(P \rightarrow \Diamond T)$
	Before R	► $\Diamond R \rightarrow (P \rightarrow (\neg R \, \mathcal{U} \, (T \wedge \neg R))) \, \mathcal{U} \, R$
	After L	► $\Box(L \rightarrow \Box(P \rightarrow \Diamond T))$
	Between L and R	► $\Box((L \wedge \neg R \wedge \Diamond R) \rightarrow (P \rightarrow (\neg R \, \mathcal{U} \, (T \wedge \neg R))) \, \mathcal{U} \, R)$
	After L Until R	► $\Box(L \wedge \neg R \rightarrow ((P \rightarrow (\neg R \, \mathcal{U} \, (T \wedge \neg R)))\mathcal{W} R)$

Before we proceed to the scopes analysis, we give a few recurring definitions in the proof sketches for this pattern.

Definition 4.5.1 — $(P \rightarrow (\neg R \, \mathcal{U} \, (T \wedge \neg R)))$. Consider a sequence σ of states $s_0, s_1, \ldots s_n$, where s_0 is the initial state and s_n is the final state of the program execution, and a position i such that $0 \leq i \leq n$. We analyze the following statement using Definition B.3.7:

$$(\sigma, i) \vDash (P \rightarrow (\neg R \, \mathcal{U} \, (T \wedge \neg R))) \text{ therefore if } (\sigma, i) \vDash P \text{ then } (\sigma, i) \vDash (\neg R \, \mathcal{U} \, (T \wedge \neg R))$$

The first part of the implication states that P holds at position i. The second part of the implication is analyzed below:

$$\begin{aligned}(\sigma, i) \vDash (\neg R \, \mathcal{U} \, (T \wedge \neg R)) \quad \text{iff} \quad & \exists k,\ i \leq k \leq n,\ \text{s.t. } (\sigma, k) \vDash T \wedge \neg R \\ & \text{and } \forall m,\ j \leq m < k,\ (\sigma, m) \vDash \neg R \text{ (Definition B.3.19)}\end{aligned}$$

This expression states that $\neg R$ holds from position i until the moment $T \wedge \neg R$ occurs. Combining the analysis results for each part of the implication, we have that the following statements must hold:

- $(\sigma, i) \vDash P$: P holds at i, $0 \leq i \leq n$, and $\neg R$ occurs at this position. R cannot occur at this position because the states/events that take place after i would not be considered in the defined scopes (Section 2.4.1).
- $(\sigma, k) \vDash T \wedge \neg R$: T and $\neg R$ hold at position k, $i \leq k \leq n$. P may occur at k, but R cannot occur because of $\neg R$ ($R \wedge \neg R$ is a contradiction (Definition B.3.15 (2))).
- $(\sigma, m) \vDash \neg R$: $\forall m,\ i \leq m < k$, $\neg R$ holds. P may occur in this interval, but T cannot occur because it has already taken place at k (Definition B.3.19). R cannot hold in this interval neither because this would lead to a contradiction, as stated previously.

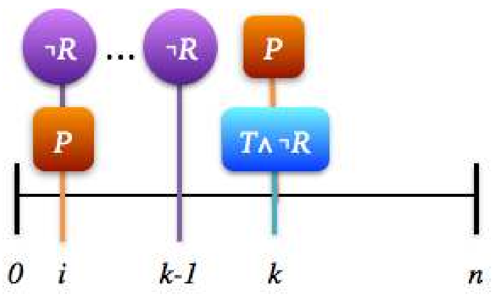


Figure 4.15: *Response pattern - Definition 4.5.1: P must hold at k and/or before k*

⇒ *The Definition 4.5.1 is used in Definition 4.5.2 and in Proof-Sketch 4.5.4.*

Definition 4.5.2 — $((P \rightarrow (\neg R\ \mathcal{U}\ (T \wedge \neg R)))\ \mathcal{U}\ R)$. Consider a sequence σ of states $s_0, s_1, \dots s_n$, where s_0 is the initial state and s_n is the final state of the program execution, and a position i such that $0 \leq i \leq n$. We apply the Definition B.3.19 and get the statement :

$$\begin{aligned}(\sigma, i) \models ((P \rightarrow (\neg R\ \mathcal{U}\ (T \wedge \neg R)))\ \mathcal{U}\ R) \quad \text{iff} \quad & \exists \nu,\ i \leq \nu \leq n,\ \text{s.t. } (\sigma, \nu) \models R \\ & \text{and } \forall h,\ i \leq h < \nu,\ (\sigma, h) \models (P \rightarrow (\neg R\ \mathcal{U}\ (T \wedge \neg R)))\end{aligned}$$

The formula $(P \rightarrow (\neg R\ \mathcal{U}\ (T \wedge \neg R)))$ has been already analyzed in Definition 4.5.1. Therefore, combining these definition results with the above statement, we have that the following statements must hold:

- $(\sigma, h) \models (P \rightarrow (\neg R\ \mathcal{U}\ (T \wedge \neg R)))$: $\forall h,\ i \leq h < \nu$, the interval defined by the statements of Definition 4.5.1 must hold one or more times.
- $(\sigma, \nu) \models R$: considering that the occurrence of $(T \wedge \neg R)$ holds at position k, $k \geq i$, we have that R occurs at some position ν, $k < \nu \leq n$. Since no states/events associated with R are considered in scopes definitions (Section 2.4.1), we consider that only R occurs at ν.

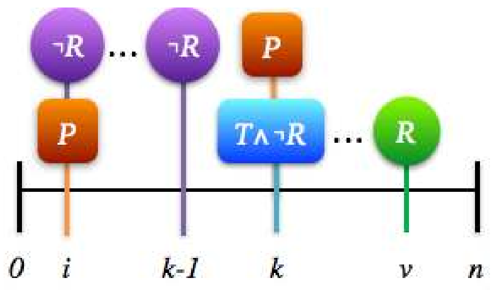


Figure 4.16: *Response pattern - Definition 4.5.2: P must hold at k and/or before k*

⇒ *The Definition 4.5.2 is used in Proof-Sketches 4.5.1, 4.5.3 and 4.5.4.*

Remark. According to Manna and Pnueli [MP92], the following formulas represent the *response* pattern (recurrence property):

- $\Box\Diamond((\neg P)\ \mathcal{B}\ T)$: this formula has the canonical form of a recurrence property (Section 3.2.2). It states that there are (infinitely) many positions in which all previous requests have been responded to. These are positions such that no new requests (represented by P) have been posted since the last response (represented by T) (Definitions B.3.18, B.3.24 and B.3.27)[MP92].
- $\Box(P \rightarrow \Diamond T)$: this formula keeps the semantics of a recurrence property. It states that for each position j, $0 \leq j \leq n$, if P holds at j there must be a later position k, $j \leq k \leq n$ satisfying T (Definitions B.3.18 and B.3.17). In other words, every position in which P holds coincides or is followed by a position in which T holds [MP92].

Therefore, we have that $\Box\Diamond((\neg P)\ \mathcal{B}\ T) \equiv \Box(P \rightarrow \Diamond T)$.

4.5.1 Before R

Proposition 4.5.1 — Recurrence Property (Response Pattern) in the Before R Scope. The *recurrence* property represented by the *response* pattern is preserved in the *before R* scope.

$$\Diamond R \rightarrow ((P \rightarrow (\neg R\ \mathcal{U}\ (T \wedge \neg R)))\ \mathcal{U}\ R)$$

Proof Sketch 4.5.1. Consider a sequence σ of states $s_0, s_1, \dots s_n$, where s_0 is the initial state and s_n is the final state of the program execution. The following statement must hold:

$$(\sigma, 0) \models \Diamond R \rightarrow ((P \rightarrow (\neg R\ \mathcal{U}\ (T \wedge \neg R)))\ \mathcal{U}\ R) \quad \text{therefore} \quad \text{if } (\sigma, 0) \models \Diamond R \text{ then } (\sigma, 0) \models ((P \rightarrow (\neg R\ \mathcal{U}\ (T \wedge \neg R)))\ \mathcal{U}\ R)$$

Using this statement, we can analyze each part of the implication as follows:

- **Case 1.** $(\sigma, 0) \models \Diamond R$ iff $(\sigma, k) \models R$ for some k, $0 \leq k \leq n$ (Definition B.3.17). This expression deals only with the occurrence of R, which limits the scope. Thus we can assume that R occurs at position k;
- **Case 2.** $(\sigma, 0) \models ((P \rightarrow (\neg R\ \mathcal{U}\ (T \wedge \neg R)))\ \mathcal{U}\ R)$. It is the Definition 4.5.2 for $i = 0$.

Analyzing both parts of the implication, we have that the first part guarantees that R occurs at k; and, the second part assures that T responds to P before the occurrence of R (Figure 4.16). Hence, we assure that T always responds to P, and the recurrence property is preserved in this scope. ■

4.5.2 After L

Proposition 4.5.2 — Recurrence Property (Response Pattern) in the After L Scope. The *recurrence* property represented by the *response* pattern is preserved in the *after L* scope.

$$\Box Q, \text{ where } Q = (L \rightarrow \Box(P \rightarrow \Diamond T))$$

Proof Sketch 4.5.2. Consider a sequence σ of states $s_0, s_1, \dots s_n$, where s_0 is the initial state and s_n is the final state of the program execution, and a position j such that $0 \leq j \leq n$. We analyze the Q statement first:

$$(\sigma, j) \models L \rightarrow \Box(P \rightarrow \Diamond T).$$

This statement says that if L occurs at a position k s.t. $j \leq k \leq n$, then $(P \rightarrow \Diamond T)$ holds $\forall m$ s.t. $k \leq m \leq n$ (Definition B.3.18). Therefore, the recurrence property is preserved in the Q statement since $(P \rightarrow \Diamond T)$ is equivalent to the canonical form of the recurrence property. Now, considering that the original formula is $\Box Q$, we have that every position in which L occurs coincides or is followed by $\Box(P \rightarrow \Diamond T)$. Hence, $\Box Q$ assures that the recurrence property represented by Q always occurs in the computation.

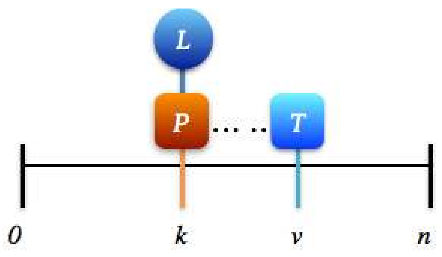


Figure 4.17: *Response pattern - After L scope*

■

4.5.3 Between L and R

Proposition 4.5.3 — Recurrence Property (Response Pattern) in the Between L and R Scope. The *recurrence* property represented by the *response* pattern is preserved in the *between L and R* scope.

$$\Box Q, \text{ where } Q = ((L \wedge \neg R \wedge \Diamond R) \rightarrow (P \rightarrow (\neg R\ \mathcal{U}\ (T \wedge \neg R)))\ \mathcal{U}\ R)$$

Proof Sketch 4.5.3. Consider a sequence σ of states $s_0, s_1, \ldots s_n$, where s_0 is the initial state and s_n is the final state of the program execution, and a position j such that $0 \leq j \leq n$. We first analyze the Q statement, using Definition B.3.7:

$$(\sigma, j) \vDash (L \wedge \neg R \wedge \Diamond R) \rightarrow ((P \rightarrow (\neg R\ \mathcal{U}\ (T \wedge \neg R)))\ \mathcal{U}\ R) \quad \text{therefore if } (\sigma, j) \vDash L \wedge \neg R \wedge \Diamond R, \text{ then } (\sigma, j) \vDash ((P \rightarrow (\neg R\ \mathcal{U}\ (T \wedge \neg R)))\ \mathcal{U}\ R)$$

Using this statement, we can analyze each part of the implication as follows:

- **Case 1.** $(\sigma, j) \vDash L \wedge \neg R \wedge \Diamond R$ iff $(\sigma, j) \vDash L$ and $(\sigma, j) \vDash \neg R$ and $(\sigma, j) \vDash \Diamond R$. This expression deals only with the occurrence of L and R, which are the states/events that limit the scope. It says that L occurs at position j and R does not. Moreover, we have that:
  - $(\sigma, j) \vDash \Diamond R$ iff $(\sigma, \nu) \vDash R$ for some $k > j$ (Definition B.3.17). The position ν cannot be equal to position j, otherwise we would have at position j the occurrence of $\neg R \wedge R$, which is a contradiction. Therefore, we can assume that R occurs at position ν, $j < \nu \leq n$.
- **Case 2.** $(\sigma, j) \vDash ((P \rightarrow (\neg R\ \mathcal{U}\ (T \wedge \neg R)))\ \mathcal{U}\ R)$. It is the Definition 4.5.2 for i, s.t. $j \leq i < \nu$, ν the position in which R occurs.

Analyzing both parts of the implication, we have that the first part guarantees that L occurs at j and R occurs at ν, $j < \nu \leq n$; and, the second part assures that T responds to P before the occurrence of R (Figure 4.16). Hence, we assure that T always responds to P, and the recurrence property is preserved in the scope defined by the Q statement.

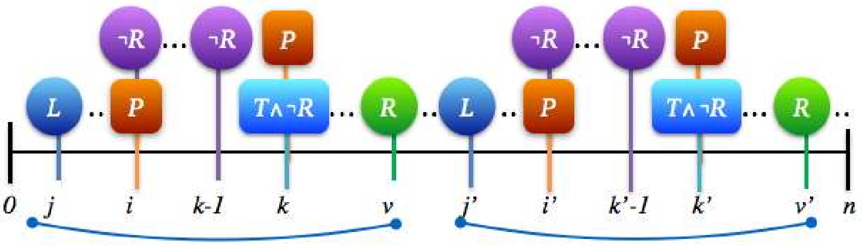


Figure 4.18: *Response pattern - Between L and R scope - Scope repetition*

Since the response pattern is represented by $\Box Q$ in this scope, we have that $(\sigma, j) \vDash \Box Q$ iff $(\sigma, l) \vDash Q$,$\forall l$, $j \leq l \leq n$ (Definition B.3.18). Hence, $\Box Q$ assures that the recurrence property represented by Q always occurs in the computation (Figure 4.18).

■

4.5.4 After L Until R

Proposition 4.5.4 — Recurrence Property (Response Pattern) in the After L Until R Scope. The *recurrence* property represented by the *response* pattern is preserved in the *after L until R* scope.

$$\Box Q, \text{ where } Q = (L \wedge \neg R \rightarrow ((P \rightarrow (\neg R\ \mathcal{U}\ (T \wedge \neg R)))\mathcal{W} R))$$

Proof Sketch 4.5.4. Consider a sequence σ of states $s_0, s_1, \ldots s_n$, where s_0 is the initial state and s_n is the final state of the program execution, and a position j such that $0 \leq j \leq n$. We first analyze the Q statement using Definition B.3.7:

$$(\sigma, j) \vDash (L \wedge \neg R) \rightarrow ((P \rightarrow (\neg R\ \mathcal{U}\ (T \wedge \neg R)))\mathcal{W} R) \quad \text{therefore} \quad \text{if } (\sigma, j) \vDash (L \wedge \neg R) \text{ then } (\sigma, j) \vDash ((P \rightarrow (\neg R\ \mathcal{U}\ (T \wedge \neg R)))\mathcal{W} R)$$

The first part of the implication states that L holds at position j and R cannot hold at this same position. The second part of the implication is a *waiting-for* expression. According to this operator definition (Definition B.3.20), we have that one of the following statements must hold:

- $\Box((P \rightarrow (\neg R\ \mathcal{U}\ (T \wedge \neg R))))$: $\forall u, j \leq u \leq n, (P \rightarrow (\neg R\ \mathcal{U}\ (T \wedge \neg R)))$ holds. This means that the statements of Definition 4.5.1 (Figure 4.15) hold in the interval $[i, n]$
- $((P \rightarrow (\neg R\ \mathcal{U}\ (T \wedge \neg R)))\mathcal{U}R)$: Definition 4.5.2 (Figure 4.16)

Thus, the Q statement says that after L, the following statements must hold:

- $((P \rightarrow (\neg R\ \mathcal{U}\ (T \wedge \neg R)))\mathcal{U}R)$, which defines that T responds to P before the occurrence of R, holds after the occurrence of L (Definition 4.5.2); or,
- $\Box((P \rightarrow (\neg R\ \mathcal{U}\ (T \wedge \neg R))))$, which defines that T responds to P and does not consider an occurrence of R in the future, holds after the occurrence of L (Definition 4.5.1).

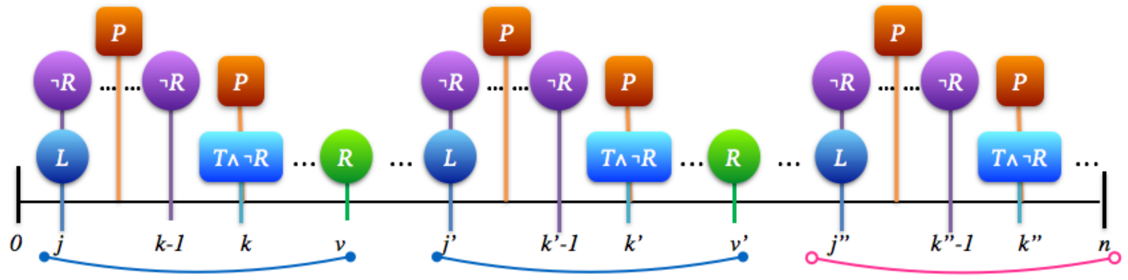


Figure 4.19: *Response pattern - After L Until R scope - Scope repetition*

Therefore, we have that Q is a recurrence property in the scope *after L until R*. And, since the response pattern is represented by $\Box Q$ in this scope, we have that $(\sigma, j) \models \Box Q$ iff $(\sigma, l) \models Q$, $\forall l,\ j \leq l \leq n$ (Definition B.3.18). Hence, $\Box Q$ assures that the recurrence property represented by Q always occurs in the computation (Figure 4.19). ■

4.6 Results

To summarize the results we obtained with this study, we present the following theorems:

Theorem 4.6.1 — Safety Property in the SPS Scopes. The *safety* property represented by the *universality*, *absence* and *precedence* patterns is preserved in all SPS scopes.

The result for the global scope can be reached directly [DAC99, MP92]. The results for the other scopes are given by Propositions 4.1.1, 4.1.2, 4.1.3 and 4.1.4 for the universality pattern; Propositions 4.2.1, 4.2.2, 4.2.3 and 4.2.4 for the absence pattern; and, Propositions 4.3.1, 4.3.2, 4.3.3 and 4.3.4 for the precedence pattern.

Theorem 4.6.2 — Guarantee Property in the SPS Scopes. The *guarantee* property represented by the *existence* pattern is preserved in all SPS scopes.

The result for the global scope can be reached directly [DAC99, MP92], and the results for the other scopes are given by Propositions 4.4.1, 4.4.2, 4.4.3 and 4.4.4.

Theorem 4.6.3 — Recurrence Property in the SPS Scopes. The *recurrence* property represented by the *response* pattern is preserved in all SPS scopes.

The result for the global scope can be reached directly [DAC99, MP92], and the results for the other scopes are given by Propositions 4.5.1, 4.5.2, 4.5.3 and 4.5.4.

4.6.1 Testable SPS Formulas

Table 4.6 presents all the SPS LTL formulas that can be used to specify testable properties. The ✓ symbol indicates that other works have already shown that the formula preserves the property (and therefore it is testable) in the given scope [NGH93, FFJ+10]; and, the ► symbol indicates the formulas that we analyzed to show that they also preserve the semantics of their respective property class/pattern.

Table 4.6: *Testable LTL formulas for Patterns and Scopes [DAC99, SGRE11]*

Pattern	Scope	SPS
Universality (P)	Global	✓ $\Box(P)$
	Before R	► $\Diamond R \rightarrow (P\ \mathcal{U}\ R)$
	After L	► $\Box(L \rightarrow \Box(P))$
	Between L and R	► $\Box((L \wedge \neg R \wedge \Diamond R) \rightarrow (P\ \mathcal{U}\ R))$
	After L Until R	► $\Box(L \wedge \neg R \rightarrow (P \mathcal{W} R))$
Absence (P)	Global	✓ $\Box(\neg P)$
	Before R	► $\Diamond R \rightarrow (\neg P\ \mathcal{U}\ R)$
	After L	► $\Box(L \rightarrow \Box(\neg P))$
	Between L and R	► $\Box((L \wedge \neg R \wedge \Diamond R) \rightarrow (\neg P\ \mathcal{U}\ R))$
	After L Until R	► $\Box(L \wedge \neg R \rightarrow (\neg P \mathcal{W} R))$
Precedence (T, P)	Global	✓ $\neg P \mathcal{W} T$
	Before R	► $\Diamond R \rightarrow (\neg P\ \mathcal{U}\ (T \vee R))$
	After L	► $\Box \neg L \vee \Diamond(L \wedge (\neg P \mathcal{W} T))$
	Between L and R	► $\Box((L \wedge \neg R \wedge \Diamond R) \rightarrow (\neg P\ \mathcal{U}\ (T \vee R)))$
	After L Until R	► $\Box(L \wedge \neg R \rightarrow (\neg P \mathcal{W}(T \vee R)))$
Existence (P)	Global	✓ $\Diamond(P)$
	Before R	► $\neg R \mathcal{W}(P \wedge \neg R)$
	After L	► $\Box(\neg L \vee \Diamond(L \wedge \Diamond P))$
	Between L and R	► $\Box(L \wedge \neg R \rightarrow (\neg R \mathcal{W}(P \wedge \neg R)))$
	After L Until R	► $\Box(L \wedge \neg R \rightarrow (\neg R\ \mathcal{U}\ (P \wedge \neg R)))$
Response (T, P)	Global	✓ $\Box(P \rightarrow \Diamond T)$
	Before R	► $\Diamond R \rightarrow (P \rightarrow (\neg R\ \mathcal{U}\ (T \wedge \neg R)))\ \mathcal{U}\ R$
	After L	► $\Box(L \rightarrow \Box(P \rightarrow \Diamond T))$
	Between L and R	► $\Box((L \wedge \neg R \wedge \Diamond R) \rightarrow (P \rightarrow (\neg R\ \mathcal{U}\ (T \wedge \neg R)))\ \mathcal{U}\ R)$
	After L Until R	► $\Box(L \wedge \neg R \rightarrow ((P \rightarrow (\neg R\ \mathcal{U}\ (T \wedge \neg R)))\mathcal{W} R)$

Recall the example of Section 2.3.1, which has three methods that perform arithmetic operations with the integer variable x: (i) `reset` assigns to the variable x the value 0; (ii) `isZero` checks whether the value of x is zero; and, (iii) `add1`, which adds 1 to the value of x. We have defined in Section 3.3.2 examples of properties in local scopes concerning to these methods (Examples 3.3.1 and 3.3.2):

Example 3.3.1. The property states that the value of x must be checked at least once after the `reset` method is called. Therefore, to specify this property, we have the *existence* pattern for `P = isZero`; and, the *after L* scope for `L = reset`. Using the formula from Table 2.3, we get the following LTL property:

$$\Box(\neg(\text{reset}) \vee \Diamond(\text{reset} \wedge \Diamond\text{isZero}))$$

Example 3.3.2 The property checks if after the `reset` method is called, the `isZero` method is called at least once until the `add1` method is called. In this case, we have the *existence* pattern for `P = isZero`; and, the *after L until R* scope for `L = reset` and `R = add1`. Using the formula from Table 2.3, we get the following specification:

$$\Box(\mathtt{reset} \wedge \neg(\mathtt{add1}) \rightarrow (\neg(\mathtt{add1})\ \mathcal{U}\ (\mathtt{isZero} \wedge \neg(\mathtt{add1}))))$$

Analyzing the examples, we can observe that the same formulas that were used to specify the properties in local scopes are presented in Tables 2.3 and 4.6. Therefore, we conclude that the properties of the above examples are testable, and we can use the corresponding LTL formulas as property specifications.

4.7 Conclusion

In this chapter, we described the process of extending the testability concepts for the SPS patterns (*universality*, *absence*, *precedence*, *existence* and *response*) in local scopes (*before R*, *after L*, *between L and R*, and *after L until R*). During this process, we prove that a broader range of SPS formulas can be used to derive testable properties, thus, we increase the possibilities of specifying properties that will provide a success/fail verdict when checked against a finite-execution program with a model checker. In addition, we give a formal validation of the SPS formulas concerning to the LTL representation, using the definitions of LTL syntax and semantics (Sections B.2 and B.3). Hence, our contribution is different from the work of Salamah et al. [SGRM05], which discusses the correctness of the SPS formulas through an empirical study using the SPIN model checker [Hol97] and several test cases.

In the next part of this thesis (Part III), we use the test purposes classification based on the testability analysis (Chapters 3 and 4) to simplify the definition of properties in a Java model checker. The model checker we chose for this purpose is the Java PathFinder (JPF) [Jav].

Remark 1. Salamah et al. [SGRM05] have proposed a new set of LTL formulas based on the SPS [Pat]. This new set is slightly different from the original SPS formulas, and according to the authors, it can generate nondeterministic Büchi automata with reduced states, which can be useful when dealing with model checking. These new formulas are implemented in the tool Prospec [MGR04], which was developed by the same research group to simplify the process of specifying properties in LTL. Hence, we refer to this new set of formulas as *Prospec formulas*.

Remark 2. Since the Prospec formulas can represent the same SPS patterns and scopes we analyzed in our properties testability study, we have analyzed the Prospec formulas to see if they can also be used to specify testable properties (and therefore, check whether they preserve the semantics of the original SPS formulas). This study is presented in Appendix C. *Nevertheless, in the remaining of our work, we refer only to the SPS original formulas* [DAC99, Pat].

III— Properties Formal Verification

5 — Defining and Checking Properties

After studying the topic of properties specification (Chapter 2), and defining a classification of testable properties (Part II), we must now apply this knowledge in practice using model checking. To perform the integration of the classification of testable properties with Java model checking, we have chosen to work with the model checker Java PathFinder (JPF) [Jav]. Although we have analyzed other Java model checkers (e.g. Bandera [Ban], Spin [SPI]), we concluded that JPF would be the best choice to use in our approach, since it can be extended with *property definitions in Java*, the same programming language of the IUT, and *listeners*, which can monitor the program execution during the verification process. A detailed description of JPF is given in Section 5.2. After choosing JPF as the model checker for our integration approach, we have to convert the specified property into a formal representation that must be readable for JPF. For this purpose, we chose to convert the property specification into a minimal deterministic automaton (DFA). JPF will then be able to read the automaton through the use of a listener, and traverse it while verifying the corresponding Java program. Our integration approach is depicted in Figure 5.1.

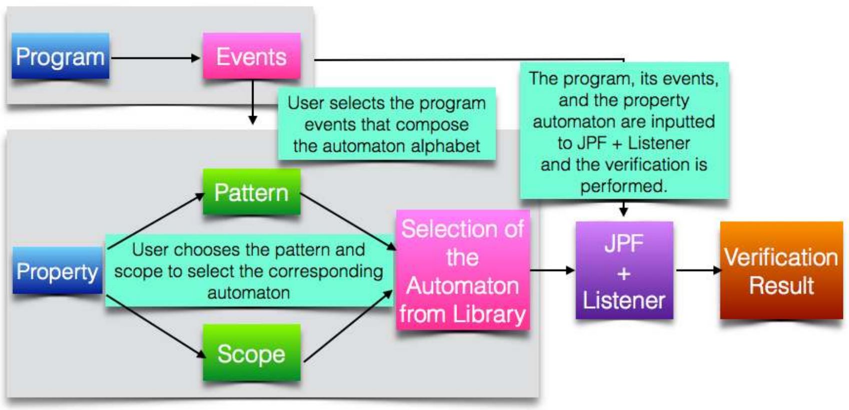

Program
Events
User selects the program events that compose the automaton alphabet
The program, its events, and the property automaton are inputted to JPF + Listener and the verification is performed.
Pattern
Property
User chooses the pattern and scope to select the corresponding automaton
Scope
Selection of the Automaton from Library
JPF + Listener
Verification Result

Figure 5.1: *Integrating testable properties specification with model checking*

The mechanisms we used to make this integration approach available for users are presented in this chapter. First, we describe the process of representing a property using DFA, and the creation of the automata library using the JFlap tool [JFL] (Section 5.1.1). Next, we give an overview of the model checker JPF and its features. The last topic of this chapter covers the implementation of a JPF listener, which is responsible for reading the property automaton and traversing it as the verification process is executed. The application of these mechanisms in the development of our tool, JPF-SPS, is presented in Chapter 6.

5.1 SPS - Automata Representation

To implement the properties in JPF, we have chosen to have an automata library for each combination of SPS pattern and scope that derives a testable property; and a JPF listener that would receive as input a property automaton and the program to perform the property verification.

Since the automaton would be used in the verification, the best representation would be the minimal state DFA. Thus, we did not have to deal with nondeterminism, and would reach a verdict by traversing a minimum number of states in the automaton. We studied several ways to convert an LTL formula to a minimal state DFA, and we did not find a reliable semi-automated way to do that. Since the SPS has several formal representations besides the LTL [Pat], we decided to use the regular expressions (Section 2.3.3, Table 2.4). This decision was based on the fact that the conversion of a regular expression to a DFA can be done by several algorithms, and this conversion can be performed in a semi-automated manner by using tools that are freely available. In our case, we used the JFLAP tool [Rod06], which is described in Section 5.1.1. The restriction that is imposed by the use of regular expressions is the fact that it can only describe program finite executions.

5.1.1 Java Formal Language and Automata Package (JFLAP)

The Java Formal Language and Automata Package (JFLAP) [Rod06] is a software that was developed as a teaching tool for formal languages. JFLAP has strong recommendations for use in theoretical courses [JFL], and its correctness has been proved in a list of publications [Rod06, JFL]. It is freely available at the website [JFL], and some of its features are: (i) interactive visualization and creation of automata; (ii) conversion of a regular expression to a nondeterministic finite automaton (NFA); (iii) conversion of an NFA to a deterministic finite automaton (DFA); (iv) conversion of a DFA to a minimal state DFA; (v) conversion of a DFA to a regular expression.

In our work, we decided to use JFLAP to generate the minimal state DFA for each combination of testable SPS pattern/scope. These automata were generated from the corresponding regular expression of testable SPS patterns/scopes. Figure 5.2 illustrates the conversion of the response pattern (global scope) DFA to its minimal state DFA.

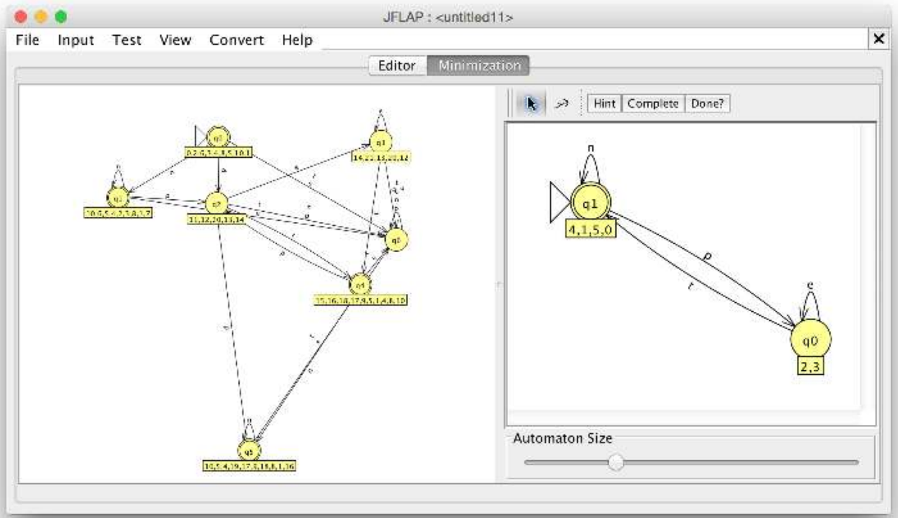


Figure 5.2: *JFLAP - Generation of a minimal state automaton for the response pattern (global scope).*

The generated automata are used in the definition of properties in JPF. Since JFlap is a highly recommended tool, as we stated previously, we assume that the automata library that was generated by JFLAP is correct.

5.1.2 Automata Library Generation

To generate the automata library, we put each formula of Table 2.4 in the JFLAP and generated the corresponding minimal state DFA. Since the regular expression only describes the behavior that satisfies the property, we have to build the complementary minimal state DFA to add the violation paths and states in the original DFA.

For instance, consider the regular expression for the *absence* pattern and *global* scope: `[-P]*`. The JFLAP generates the automaton in Figure 5.3, where the state 0 is the accepting state and it is represented by a circle with a thick border.

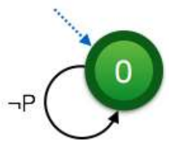


Figure 5.3: *Absence pattern/Global scope: automaton that satisfies the property.*

To add the violation paths, we completed the possible inputs to the automaton. In this case, we added a state 1 that is reached when `P` occurs (`P` is the input that is missing in the generated minimal DFA from Figure 5.3). The resulting automaton is presented in Figure 5.4.

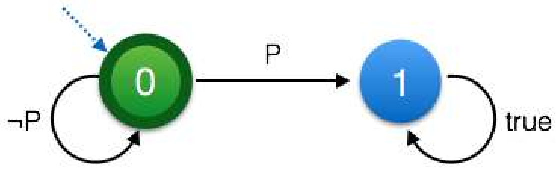


Figure 5.4: *Absence pattern/Global scope: automaton that satisfies the property with complete inputs.*

After completing the automaton, we built the complement of the DFA, as illustrated in Figure 5.5. The rejecting state is the state 1, which is represented by a circle with a dashed border.

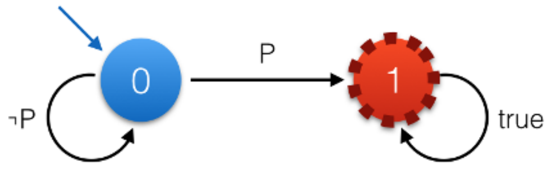


Figure 5.5: *Absence pattern/Global scope: automaton that rejects the property.*

Combining the automata from Figure 5.3 and 5.5, we generated the automaton that is depicted in Figure 5.6. This automaton has both accepting and rejecting states, and consequently, it can be traversed by JPF and reach either a success verdict or a fail verdict.

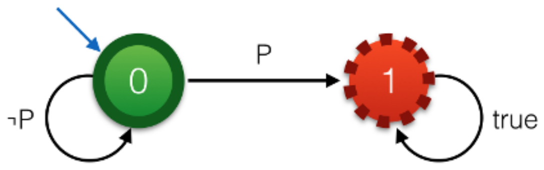


Figure 5.6: *Absence pattern/Global scope: automaton with accepting and rejecting states.*

The same process was applied to each SPS pattern/scope. Consider another example, the regular expression for the *existence* pattern and *after L* scope: `[- L]*; (L; [- P]*; P; .*)?`. The automaton that is generated by JFLAP is presented in Figure 5.7.

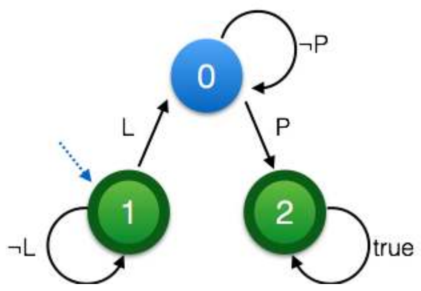


Figure 5.7: *Existence pattern/After scope: automaton automaton that satisfies the property.*

In this case, the generated DFA has all the possible inputs already. Then, we built the complement of the automaton, and generated the automaton with both accepting and rejecting states (Figure 5.8).

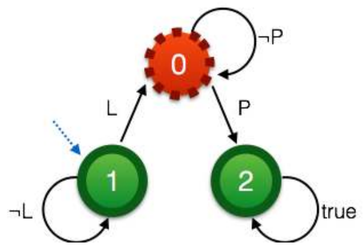


Figure 5.8: *Existence pattern/After scope: automaton with accepting and rejecting states.*

Concerning to the correctness of the generated automata, we can state that they represent the behavior that is described by the corresponding regular expression. The automata representation was mainly generated by the JFLAP tool [Rod06], which has strong recommendations for use in theoretical courses [JFL, Rod06]. And the addition of property violation paths using the complement of the generated automaton and missing events in the DFA is trivial. Thus, we can conclude that the automata library is indeed correct.

⇒ The complete collection of SPS automata with accepting and rejecting states is presented in Appendix D.

5.1.3 Automata Representation - Files

To store the generated automata in a library, we created two files with the extension `*.jas`[1], one for the nodes, and another one for the edges. The structure of the files are defined as follows:

- **File "nodes.jas"**: each line has the form: *<int nodeID>, <boolean isInitialState>, <boolean isAcceptingState>, <boolean isRejectingState>*
- **File "edges.jas"**: each line has the form: *<int nodeIDsource>, <int nodeIDtarget>, <String label>*

For instance, if we consider the DFA for the absence pattern and global scope (Figure 5.6), we have the following files:

Nodes File:
```
0, true, true, false
1, false, false, true
```

[1]JAS stands for **J**PF-SPS **A**utomaton **S**pecification.

Edges File *(the negation is represented by the symbol '!')*
```
0, 0, !P
0, 1, P
1, 1, true
```

In addition, to customize the automaton to represent a property regarding a program, we need an events file to map the default labels P, T, L, R, to the corresponding program events. The file also has the extension `*.jas`, and it has the structure:

- **File "events.jas"**: each line has the form: <*String automatonLabel*>, *String* <*eventName*>

To represent the program events that must be in the automaton alphabet, although they do not match any of the default labels, we created a new label 'NA'. An important remark is that the program event must be a single Java method name. We have this restriction to simplify the modeling of a program property to a DFA.

5.2 Java PathFinder (JPF)

Java PathFinder or JPF [Jav, VHB+03] is a virtual machine (VM) that receives the Java source code and its corresponding bytecode to analyze the execution of the program and perform the model checking. It is developed at NASA Ames Research Center and is freely available since 2005.

5.2.1 JPF Main Features

JPF is a Java application that runs on top of the host VM (usually the Java Virtual Machine). It is itself a VM that executes a Java system to find defects and check properties. Figure 5.9 presents a schematic view of these layers.

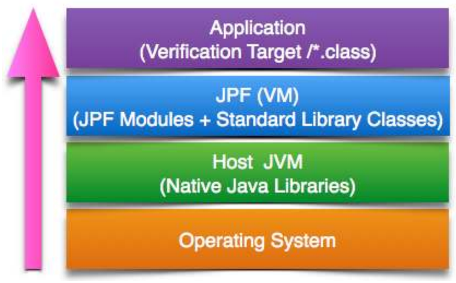

Application
(Verification Target /*.class)
JPF (VM)
(JPF Modules + Standard Library Classes)
Host JVM
(Native Java Libraries)
Operating System

Figure 5.9: *JPF components, adapted from [Jav]*

The JPF model checker receives as input a Java source code and its respective bytecode. The system properties can be written in Java and set to be checked in the configuration file (a `*.jpf` file) of JPF. After configuring the JPF, we can run the tool to check the system properties. If any violation is found, it returns program traces that indicates where the error is located. Figure 5.10 presents a diagram with the verification process in JPF.

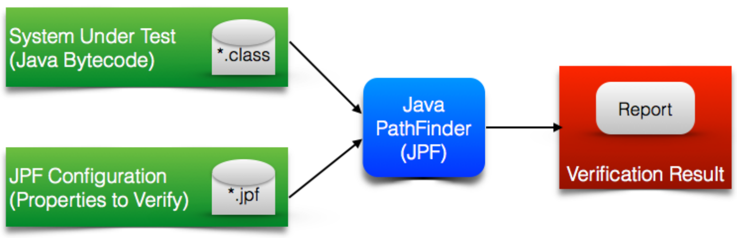


Figure 5.10: *Java code verification in JPF, adapted from [Jav]*

To avoid the problem of state explosion during the model checking process, JPF employs on-the-fly partial order reduction, a mechanism in which only operations that affect other program threads are considered during the exhaustive verification. Furthermore, JPF allows users to select one of the predefined search heuristics (e.g. depth-first search (DFS),the breadth-first search (BFS)) and configure the search according to their requirements (e.g. limit the search depth when the DFS is selected, or choose how JPF should deal when errors occur during the search). Concerning to search properties, JPF has the implementation of two properties, which we call *native JPF properties*:

- *NotDeadlockedProperty*, which verifies the occurrence of deadlock in the program.
- *NoUncaughtExceptionsProperty*, which verifies the raise of exceptions that are not caught and handled properly.

JPF also can be extended to include other search properties. Other JPF resources, like search and VM listeners, allow developers to monitor the verification process. All these resources make JPF a highly flexible tool, which can be extended and configured by users to accomplish their particular needs regarding program verification.

5.2.2 Checking Properties Automata with JPF

To make JPF capable of traversing the properties automata as it executes the program verification, we wrote a listener (the partial code of the listener is presented in Listing 5.1). This listener inherits from the JPF class *Property Listener Adapter* and implements the methods to keep control of the automaton traversing and the JPF search for property violations. It receives as inputs the program information (src path, class path, main class), the property automaton and the alphabet of events. The JPF verification starts and the listener monitors the occurrence of events of interest and traverse the automaton accordingly. For instance, suppose that we want to verify the existence of P in the scope after L (Figure 5.8) and we have the following trace of the program: T $\rightarrow$ P $\rightarrow$ L, where P, T and L are program events (methods). When the verification starts, the listener will check which edge corresponds to the event T in the automaton and then proceed from the initial node (node 1) to another node. In this case, the automaton remains in node 1, and the process is repeated for event P. Once more, the listener remains in node 1. The next event of the trace is L, which leads the listener to traverse the automaton from node 1 to node 0. Since we reached the end of the trace, and the node 0 is a rejecting state, the verification stops and reports the property violation.

To control the automaton traversing during program verification, the listener checks when a trace analysis is finished in the `stateAdvanced` method, and backtracks the automaton in the `stateBacktracked` method to continue the analysis of other program traces (to avoid problems during the analysis of other traces, we disabled the *state-matching* feature in JPF). When the verification process of searching for a property violation is interrupted due to a search limit, or a native JPF property violation, the `stateAdvanced` method is not called. In this situation, the method that is responsible for analyzing the program trace data and property verdict until the moment of interruption is the `searchFinished` method. **In all cases, a property *true* verdict is only achieved when all program traces end at an automaton accepting node; otherwise, a *fail* verdict is reported.**

Besides the verdict, the listener stores and prints the program traces and the automaton traversed edges. This information is reported to the user in a graphical form to help in a posterior analysis of the program behavior. As an example, considering again the verification of the existence of P in the scope after L (Figure 5.8) for a program that presents only the trace T $\rightarrow$ P $\rightarrow$ L, the verification report would exhibit the *fail* verdict, the analyzed trace, and the coverage of 2 out of 5 automaton edges (coverage of 40% automaton edges).

$\Rightarrow$ *For complete examples with screenshots of the developed tool, JPF-SPS, please refer to Section 6.2.*

Listing 5.1: *Implemented JPF Listener - Partial Code*

```
public class SPSListener extends PropertyListenerAdapter {
  (...)
  //Variables Initialization
  (...)
  //This method reads the events file and return a vector
  //containing the events
  private static Vector<String> getEventsFromFile (Path filePathEvents) {(...)}

  //This method reads the nodes and edges files of the selected
  //automaton and return the automaton as a vector of edges
  private static Vector<Edge> buildGraphFromFile (Path filePathNodes, Path
      filePathEdges) {(...)}

  //This is the method responsible for starting the JPF search for
  //property violations. The variables that are used in the verification
  //are initialized here.
  @SuppressWarnings("unchecked")
  @Override
  public void searchStarted(Search search) {(...)}

  //This method is responsible for checking a program instruction,
  //and traverse the property automaton accordingly.
  @Override
  public void instructionExecuted(VM vm, ThreadInfo currentThread, Instruction
      nextInstruction, Instruction executedInstruction) {
    MethodInfo mi = executedInstruction.getMethodInfo();
    String methodName = mi.getName();
    //We only consider the events that are defined in the
    //events file (automaton alphabet)
    if (isProgramMethod (methodName, programEvents)) {
      //If the executed instruction is a repeated instruction,
      //we can skip it since the next automaton state would not change
      if(executedInstruction.isFirstInstruction()) {
        pInstructions.add(methodName);
        currentTrace = currentTrace.concat(methodName+"\n");
        currentCompleteTrace = currentCompleteTrace.concat(methodName + "\n");
        //We check the edge label that matches the method
        Edge e = isMethodInEdge(nextTransitions, methodName);
        if (e != null) {
          //Here we keep track of the traces and the
          //traversed automaton edges
          currentTraceEdges.add(e);
          traceEdges = traceEdges.concat(e.toString() + "\n");
          if (!e.isFlag()) {
            traversedEdges.add(e);
            nonTraversedEdges.remove(e);
          }
          e.setFlag(true);
          (...)
          //Now we advance the automaton
          currentNode = e.getTarget();
          nextTransitions = getNextTransitions(currentNode.getId());
        }
      }
    }
    else {
      //Here we keep track of the events that are executed by JPF
      //but do not belong to the automaton alphabet. This information
      //can be useful for coverage analysis.
    }
  }

  //This method is called after JPF advances in the search.
  @Override
  public void stateAdvanced(Search search) {
    if (search.isEndState()) {
      //We finished the analysis of a program trace.
      collectCoverageData(false);
    }
  }
  //This method backtracks the automaton to continue the verification.
  @Override
  public void stateBacktracked(Search search) {(...)}

  //When the search terminates, we print the verdict.
  @Override
  public void searchFinished(Search search) {
    if (!currentTrace.isEmpty()) {
      //the search was interrupted
      collectCoverageData(true);
    }
    //printing the property verdict and coverage data
    System.out.println(printVerdict());
    System.out.println(printTraces());
    System.out.println(printTraversedEdges());
    System.out.println(printNotAdvancingTraces());
  }
  //This method is responsible for collecting property coverage data.
  //It receives a boolean variable as argument, to determine whether
  //the data corresponds to a search that was interrupted or not.
  private void collectCoverageData(boolean interruptedSearch) {
    if (advance == false) {
      //identifying the trace that did not advance the automaton
      notAdvancingTraces.add(traceId);
    }
    //collecting program traces and coverage data
    traces.add(currentTrace);
    completeTraces.add(currentCompleteTrace);
    tracesEdges.add(traceEdges);
    prettyPrintTraces.add(prettyPrintTrace);

    if (!interruptedSearch) {
      //if the search was not interrupted, the 'currentNode' is
      //analyzed to reach a property verdict
      propertyVerdict(currentNode);
    }
    else {
      //if the search was interrupted, the last node of the automaton
      //that was reached before the interruption is analyzed for a
      //property verdict
      propertyVerdict(currentTraceEdges.lastElement().getTarget());
    }
    //Resetting variables
    (...)
  }
  //This method is responsible for determining the property verdict
  private void propertyVerdict(Node node) {
    if (node.isAcceptingState()) {
      if (!verificationResult.equals("FALSE")) {
        verificationResult = "TRUE";
      }
      else {
        //current node corresponds to a violation
        verificationResult = "FALSE";
        errorTraces.add(traceId);
      }
  }
```

```
    //Auxiliary methods for printing program traces
    //and coverage information.
    (...)
}
```

Observing Listing 5.1, we can see that the JPF listener implementation is straightforward, and it is correct by construct. This listener makes JPF able to read the property DFA and traverse it during the verification of a program, giving to the user not only the property verdict in the verification report, but also useful coverage information regarding the program and property automaton.

5.3 Conclusion

In this chapter, we presented the approach we adopted to integrate testable properties specification with the model checking of Java programs (Figure 5.1). For this purpose, we have chosen to work with Java PathFinder (JPF) [Jav], a virtual machine that works on top of JVM, and can be used to perform model checking in Java programs. JPF has several features, which allow the extension of the tool by implementing system properties in Java, and listeners to monitor points of interest during the verification of a Java program.

To apply the proposed approach in practice, it was also required to convert the testable property specification into a formal representation. The formal representation we decided to use is the minimal DFA, since it can be used to monitor the verification through JPF (Section 5.1.2). Therefore, we generated a DFA library from SPS regular expressions (Table 2.4). Then, having the program code and the testable property specification with its corresponding SPS pattern and scope, we can select the property automaton from the library and generate the property automaton. This property automaton is inputted to JPF. To make JPF able to read the automaton and traverse it while performing the program verification, a listener was implemented (Section 5.2.2). Using this listener, and having the property automaton as input, JPF can perform the verification of the property against the Java program. The result of the verification is presented in the JPF verification report, which includes data like the property verification verdict, and the coverage of the property automaton and program code.

In the next chapter (Chapter 6), we present how we used the DFA library and the JPF listener in the development of our tool, JPF-SPS. The DFA library and the JPF listener which were described here are key mechanisms that enabled us to build the JPF-SPS.

6 — JPF-SPS

Having the properties automata library and the JPF listener, we were able to build a tool that provides a friendly Graphic User Interface (GUI) to define a property and the events of interest, and set up the JPF to perform the verification of the property. The tool is called *JPF-SPS*.

To present the tool, we give an overview of its main features and interface, providing a clear step-by-step guide of using JPF-SPS to define and verify properties of a Java program. To evaluate the use of JPF-SPS, and detect its strengths and weaknesses, we applied it to a set of examples (case studies). This set of examples is composed by different types of Java systems, since we wanted to assess the behavior or JPF-SPS when defining and verifying properties against different Java programs. To conclude this chapter, we discuss the advantages of using JPF-SPS and the differences and similarities between JPF-SPS and related work.

6.1 JPF-SPS Overview

Aiming at the simplification of properties specification and verification, we built the tool JPF-SPS. Figure 6.1 presents the main components of the tool.

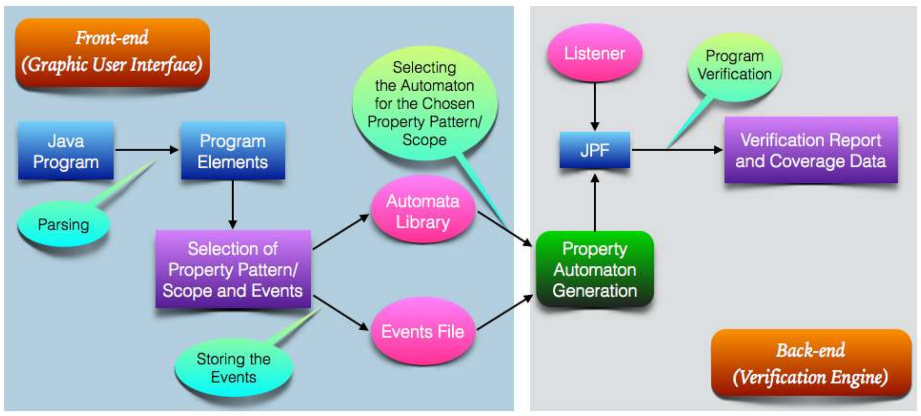

Front-end
(Graphic User Interface)
Java Program
Program Elements
Parsing
Selecting the Automaton for the Chosen Property Pattern/ Scope
Selection of Property Pattern/ Scope and Events
Automata Library
Events File
Storing the Events
Listener
Program Verification
JPF
Verification Report and Coverage Data
Property Automaton Generation
Back-end
(Verification Engine)

Figure 6.1: *JPF-SPS components diagram*

The *front-end* of the tool is composed by a GUI. Using this interface, we can input a Java program that is parsed by JPF-SPS in order to get the program elements. The program parsing is done with the help of the framework Recoder [Rec]. Having the program elements available, we can then select the property pattern and scope and drag the corresponding events for property specification.

To generate the property automaton representation (minimal DFA), JPF-SPS uses the information of which property pattern and scope were selected and choose the general automaton representation from the automata library (Section 5.1.2). To customize it according to the property specification, JPF-SPS replaces the labels of the automaton with the selected events, which were stored in a file. The DFA with the events labels is the property automaton representation that is going to be read by JPF model checker [Jav].

JPF model checker is part of the *back-end* of JPF-SPS, the verification engine. To make JPF read the property automaton and traverse it during the program verification, JPF-SPS sets the listener we implemented (Section 5.2.2) to work with the model checker. Then, when we want to verify the property against the program, JPF-SPS inputs the property automaton to JPF (plus the listener), and it performs the verification of the property while collecting information concerning to the property automaton traces coverage and program instructions coverage. The verification report containing the property automaton view, the verdict and the collected data is presented when JPF finishes the verification process.

Next, we summarize the main features of JPF-SPS. The following section presents a mini-tutorial to illustrate how we can use the tool to define and verify a property of a Java program.

JPF-SPS Main Features:

- Analysis of the Java program source code and identification of program events through a parsing process with the Recoder framework [Rec].
- Definition of a property using an interface based on the SPS. The events of interest should be selected when specifying the property pattern and scope.
- Verification of the property using the JPF and the implemented listener (Section 5.2.2). Using the property pattern, scope and events information, the property automaton is generated and traversed during the verification process by JPF. The verification verdict, property and program coverage information, and property automaton view, are reported to the user using the provided JPF-SPS interface.

6.1.1 JPF-SPS Tutorial

To illustrate how we can use the JPF-SPS, let's recall the simple arithmetic example of Section 2.3.1. The complete code of this example is presented in Listing 6.1.

Listing 6.1: *Arithmetic Example*

```java
public class ArithmeticExample {
        //adds 1 unit to the integer x
        public void add1 (int x) {
                x = x + 1;
        }
        //sets the integer x to 0
        public void reset (int x) {
                x = 0;
        }
        //checks whether the integer x has value 0
        public boolean isZero (int x) {
                return x == 0;
        }
        //main method
        public static void main(String[] args) {
                ArithmeticExample ae = new ArithmeticExample();
                int x = 5;
                ae.reset (x);
                ae.add1 (x);
                ae.isZero (x);
        }
}
```

Recalling the property we specified using SPS in Section 2.4.4:

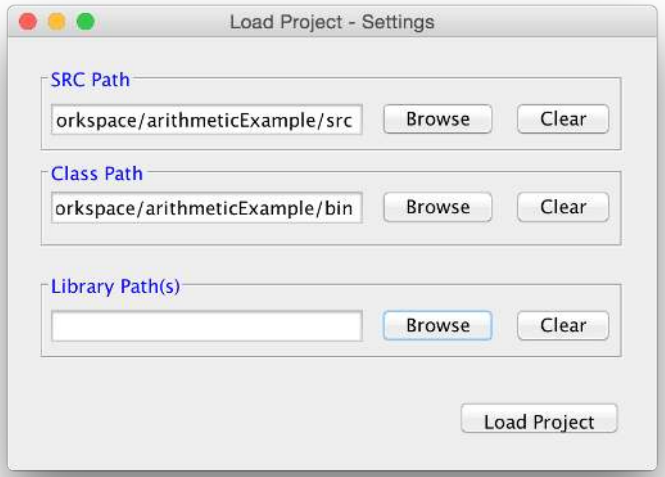
Property: *the value of* x *is checked at least once after the* reset *method is called.* To specify this property, we have the *existence* pattern for `P = isZero`; and, the *after L* scope for `L = reset`.

To define and verify the above property of *Arithmetic Example* program, we use the JPF-SPS. When we run the JPF-SPS, the first step is to provide the paths regarding the Java program.

JPF-SPS Project Settings. To start the analysis of the program, JPF-SPS requires the following program paths:

- Path to the source code: it is the path to the `*.java` files.
- Path to the bytecode: it is the path to the `*.class` files (*bytecode*).
- Path to the library folder: if the program makes use of a library of external `*.jar` files, the path to these files (e.g. *lib* folder) should be informed.

In this example, we inform the paths to the source code and to the bytecode of the *Arithmetic Example* program (Figure 6.2).



SRC Path: orkspace/arithmeticExample/src

Class Path: orkspace/arithmeticExample/bin

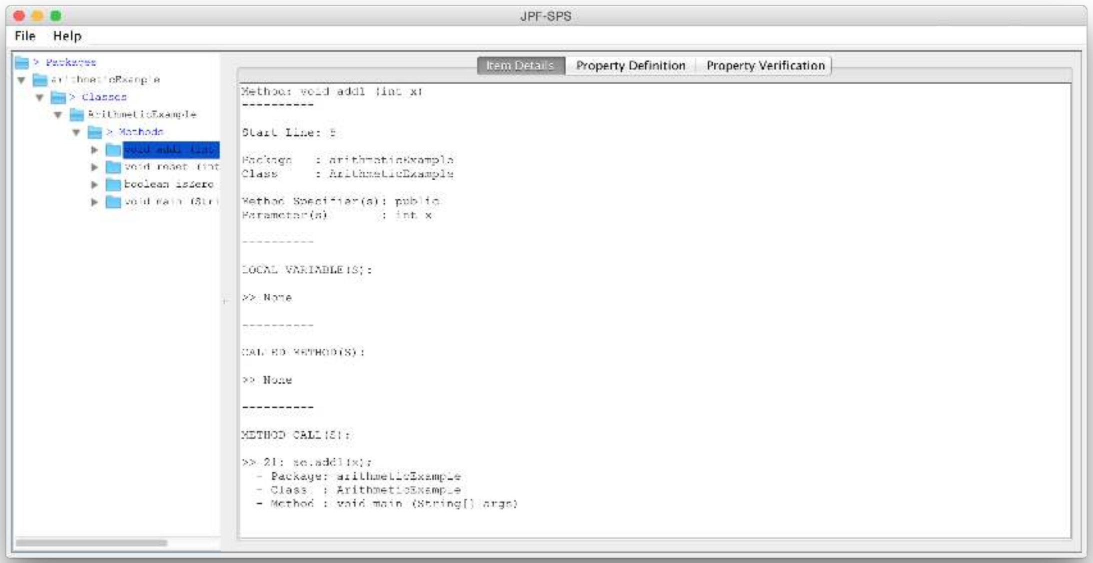
Figure 6.2: *Screenshot: Project Settings*

After we enter the paths to the Java program, JPF-SPS parses the code and present the program elements to the user in a tree on the left-hand side of the interface (Figure 6.3). On the right-hand side, there are three tabs. These tabs provide important features of JPF-SPS to the users.

JPF-SPS Tabs. Each tab provides its own features to assist users in the definition and verification of the properties.

- The first tab, *"Item Details"*, presents data regarding a selected element in the tree.
- The second tab, *"Property Definition"*, provides an intuitive interface to define a property using the SPS patterns and scopes that can specify testable properties.
- The third tab, "Property Verification", provides to the user access to the verification engine of SPS, which makes use of JPF.

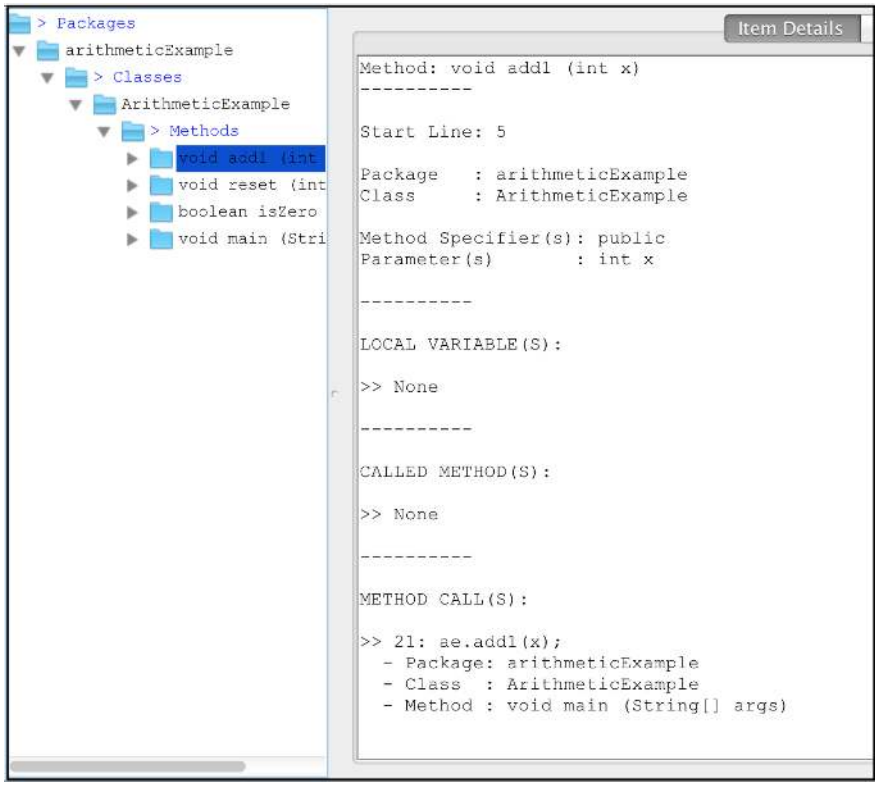
Figure 6.3 presents the tree of program elements of the *Arithmetic Example* program on the left-hand side of the interface. On the right-hand side, we can view the first tab, *"Item Details"*, when the method `add1` is selected. On this tab we can see some data concerning to this method, like the class and package it belongs to, and its parameters and local variables.



```
> Packages
▼ arithmeticExample
  ▼ > Classes
    ▼ ArithmeticExample
      ▼ > Methods
        ▶ void add1 (int
        ▶ void reset (int
        ▶ boolean isZero
        ▶ void main (Stri
```

```
Method: void add1 (int x)
----------

Start Line: 5

Package   : arithmeticExample
Class     : ArithmeticExample

Method Specifier(s): public
Parameter(s)       : int x

----------

LOCAL VARIABLE(S):

>> None

----------

CALLED METHOD(S):

>> None

----------

METHOD CALL(S):

>> 21: ae.add1(x);
  - Package: arithmeticExample
  - Class  : ArithmeticExample
  - Method : void main (String[] args)
```

(a) Full Screen



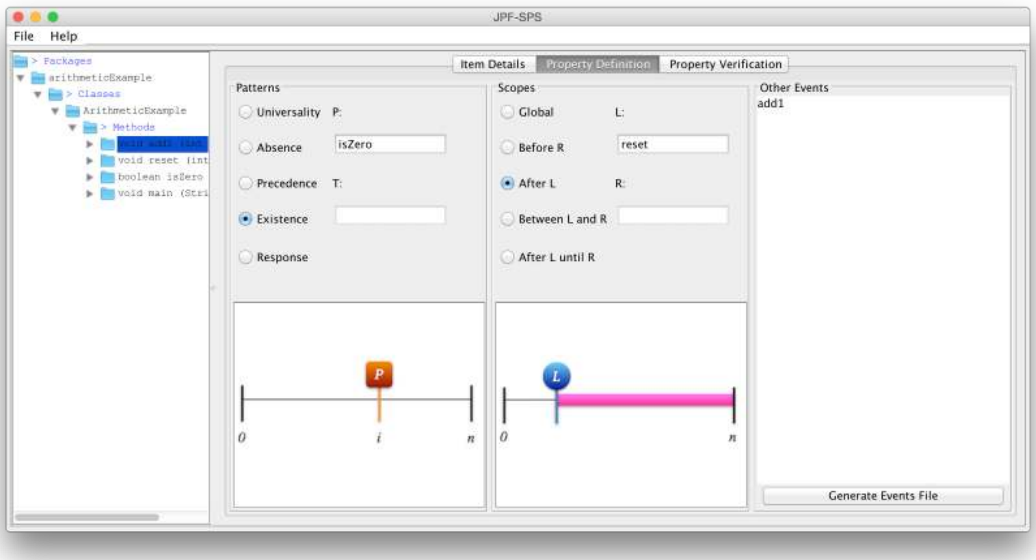
```
> Packages
▼ arithmeticExample
  ▼ > Classes
    ▼ ArithmeticExample
      ▼ > Methods
        ▶ void add1 (int
        ▶ void reset (int
        ▶ boolean isZero
        ▶ void main (Stri
```

```
Method: void add1 (int x)
----------

Start Line: 5

Package   : arithmeticExample
Class     : ArithmeticExample

Method Specifier(s): public
Parameter(s)       : int x

----------

LOCAL VARIABLE(S):

>> None

----------

CALLED METHOD(S):

>> None

----------

METHOD CALL(S):

>> 21: ae.add1(x);
  - Package: arithmeticExample
  - Class  : ArithmeticExample
  - Method : void main (String[] args)
```

(b) Detail of the Screen

Figure 6.3: *Screenshot: Item details - Method* `add1`

To specify the property we want in this example, we go to the JPF-SPS second tab, "Property Definition" (Figure 6.4), where we select the pattern, scope and events of the property. To aid the users, we also provide in this tab figures that illustrate the selected pattern and scope. For this example, we want the existence of `isZero` after `reset`. Hence, we select the existence pattern and drag the element `isZero` from the tree and drop it to the P field; and, we select the after scope and drag the element `reset` from the tree and drop it to the L field. In addition, we drag and drop the event *add1* to the *Other Events* field since it is an event of interest and, therefore, part of the automaton alphabet.

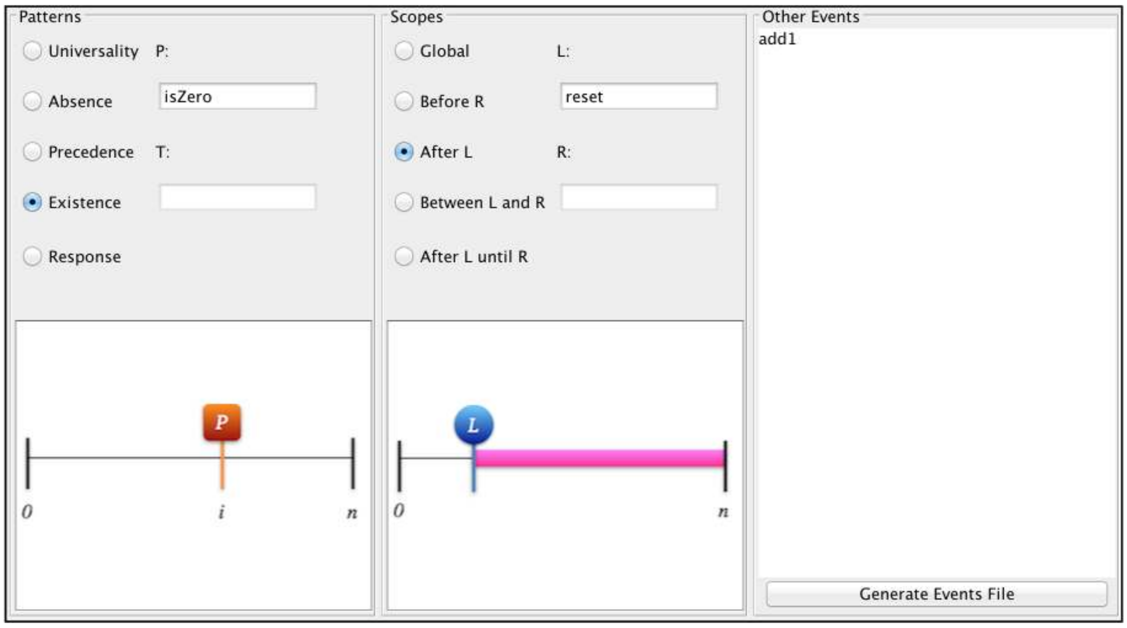


Patterns: ○ Universality P: · ○ Absence isZero · ○ Precedence T: · ● Existence · ○ Response

Scopes: ○ Global L: · ○ Before R reset · ● After L R: · ○ Between L and R · ○ After L until R

Other Events: add1

(a) Full Screen



Patterns: ○ Universality P: · ○ Absence isZero · ○ Precedence T: · ● Existence · ○ Response

Scopes: ○ Global L: · ○ Before R reset · ● After L R: · ○ Between L and R · ○ After L until R

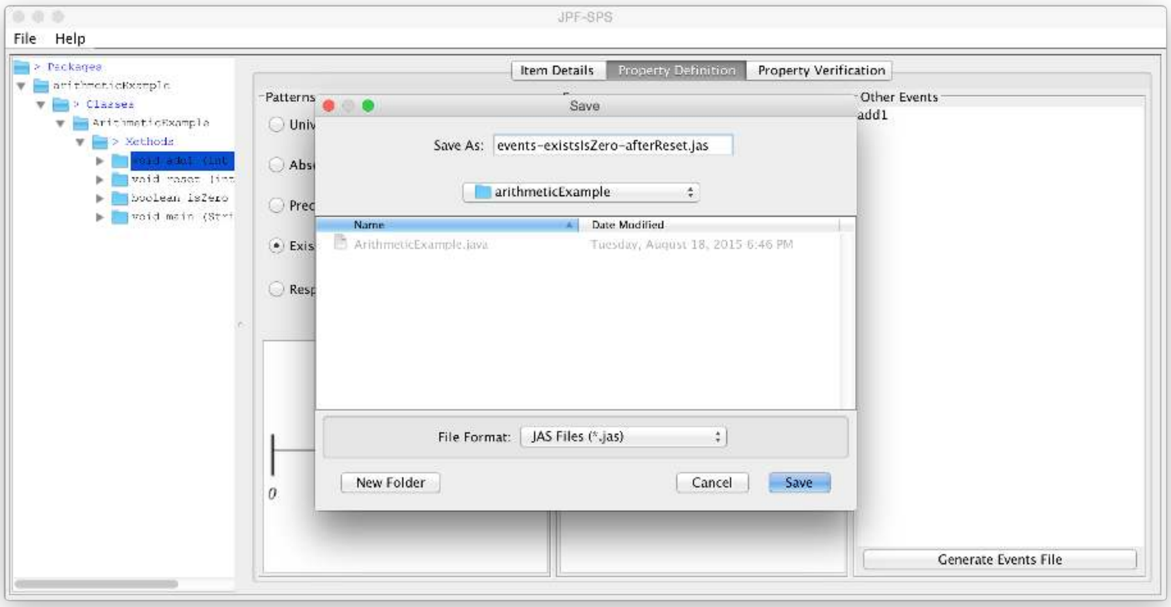
Other Events: add1

(b) Detail of the Screen

Figure 6.4: *Screenshot: Property Definition - Existence of* `isZero` *after* `reset`

After selecting the events, we click on the button to generate the mapping file that will replace the default automaton labels with the selected events (Figure 6.5). Since *add1* does **n**ot belong to the property **a**utomaton, it is marked with the label **NA**. The generated file has the following content:

File: events-existsIsZero-afterReset.jas
```
P, isZero
L, reset
NA, add1
```



Save As: events-existsIsZero-afterReset.jas

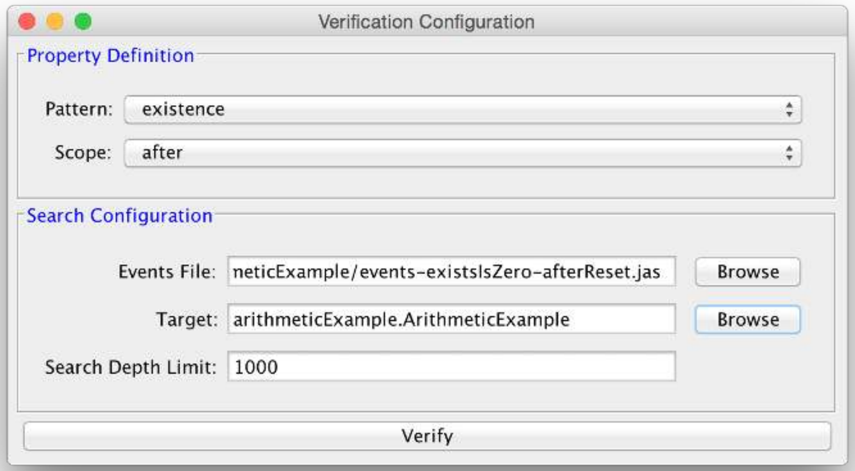
arithmeticExample

Name	Date Modified
ArithmeticExample.java	Tuesday, August 18, 2015 6:46 PM

File Format: JAS Files (*.jas)

Figure 6.5: *Screenshot: Saving the events of the property in file: events-existsIsZero-afterReset.jas*

To perform the property verification, we go to the third tab of JPF-SPS, "Property Verification", which provides to the user the *"Run Verification"* button to perform the verification of the defined property with JPF. When the user clicks on that button, a verification configuration window appears (Figure 6.6).



Pattern: existence

Scope: after

Events File: neticExample/events-existsIsZero-afterReset.jas

Target: arithmeticExample.ArithmeticExample

Search Depth Limit: 1000

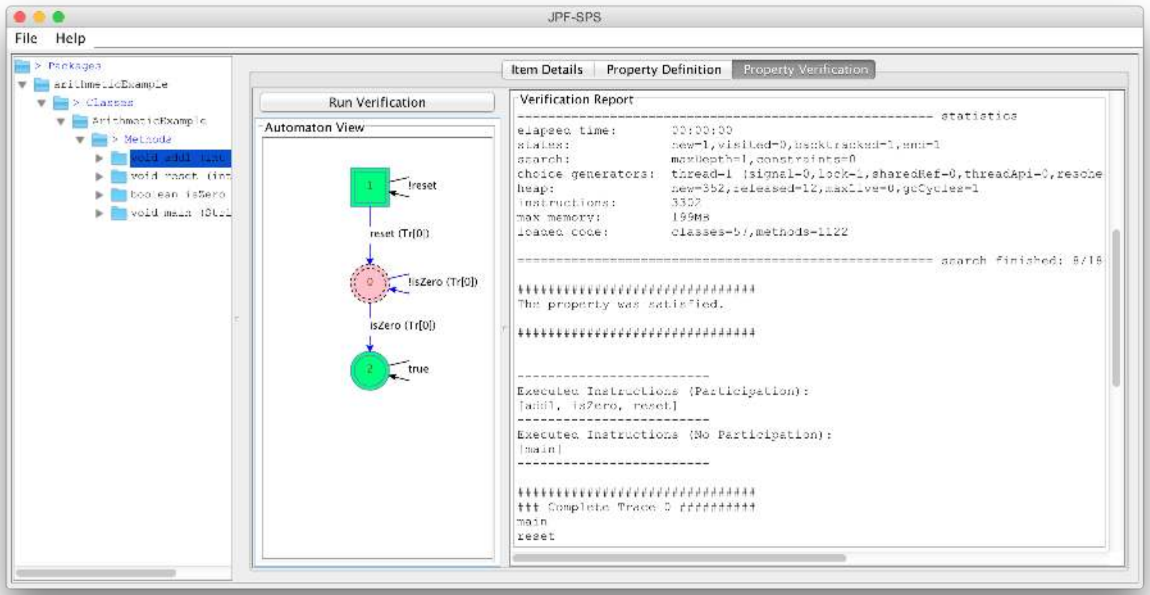
Figure 6.6: *Screenshot: Verification Configuration*

Verification Configuration. To configure the verification, we have to inform:

- The property pattern and scope. If the user has defined the property on the second tab of JPF-SPS, the pattern and scope are already set by JPF-SPS; otherwise, the user should select the corresponding pattern and scope of the property to be verified.

- The events file. If the user has defined the property on the second tab of JPF-SPS, the path to this file is already set by JPF-SPS; otherwise, the user should select the path to the corresponding file.
- The JPF target, which is the Java class of the program that contains the method `main`.
- The search depth limit. This limit is used by JPF to limit the depth-first search it performs during verification. It is particularly used when JPF-SPS needs to handle loops and recursion in the program, since we disabled the state-matching feature for the sake of program traces analysis. The default value we set is 1000. This value can be increased or decreased according to the user needs and hardware limits.

In this example, we select the *existence* pattern and *after* scope, the events file *events-existsIsZero-afterReset.jas*, and the target *ArithmeticExample* class. We kept the search limit as 1000. Then, after configuring the verification, the JPF checks the selected property. A verification report and a view of the property automaton with the new labels is also provided (Figure 6.7).



```
------------------------------------------------------------ statistics
elapsed time:          00:00:00
states:                new=1,visited=0,backtracked=1,end=1
search:                maxDepth=1,constraints=0
choice generators:     thread=1 (signal=0,lock=1,sharedRef=0,threadApi=0,resche
heap:                  new=352,released=12,maxLive=0,gcCycles=1
instructions:          3307
max memory:            199MB
loaded code:           classes=57,methods=1122

============================================================ search finished: 8/18

################################
The property was satisfied.

################################


--------------------------
Executed Instructions (Participation):
[add1, isZero, reset]
--------------------------
Executed Instructions (No Participation):
[main]
--------------------------

################################
### Complete Trace 0 ##########
main
reset
```

Figure 6.7: *Screenshot: Property Verification*

The automaton states and edges can be freely arranged by the user. Figure 6.8 presents the same automaton of Figure 6.7, but in another arrangement. The initial state of the automaton is represented by a rectangle, and the others are represented by circles. Accepting states have double line borders, and rejecting states double dashed line borders. The symbol '!' represents the logical negation. The edges that were traversed during verification are highlighted in the automaton, and the program traces that traversed an edge are identified in the label by *"Tr[<trace(s) id>]"*.

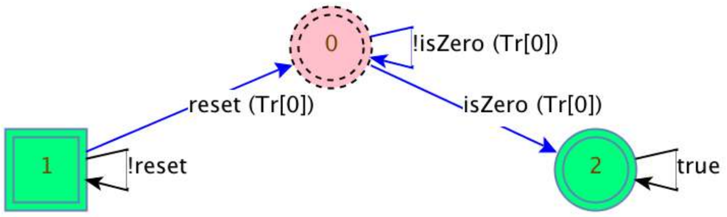


Figure 6.8: *Property Verification - Automaton View*

The verification report (Listing 6.2) presents to the user the property verification verdict, program traces and the automaton edges that were traversed during verification. In this example, the property was satisfied (there is an occurrence of `isZero` after `reset`). Three out of five edges of the automaton were covered (60%), and the program trace is:

$$\texttt{reset} \rightarrow \texttt{add1} \rightarrow \texttt{isZero}$$

Listing 6.2: *Arithmetic Example - Partial verification report*

```
JavaPathfinder core system v8.0 (rev 2+) - (C) 2005-2014 United States Government.
All rights reserved.
======================================= system under test
arithmeticExample.ArithmeticExample.main()
======================================= search started: 8/18/15 7:06 PM
(...)
======================================= search finished: 8/18/15 7:06 PM
###############################
The property was satisfied.
###############################
-------------------------
Executed Instructions (Participation):
[add1, isZero, reset]
-------------------------
Executed Instructions (No Participation):
[main]
-------------------------
###############################
### Complete Trace 0 ##########
main
reset
add1
isZero
######################
### Trace 0 ##########
reset
add1
isZero

--- Traversed Edges in this Trace: -----
(Node 1, reset / reset, Node 0)
(Node 0, add1 / !isZero, Node 0)
(Node 0, isZero / isZero, Node 2)
#########################################
### Traversed Automaton Edges  ##########
(Node 1, reset, Node 0)
(Node 0, !isZero, Node 0)
(Node 0, isZero, Node 2)
#########################################
### Non-Traversed Automaton Edges  ##########
(Node 1, !reset, Node 1)
(Node 2, true, Node 2)

### Edges Coverage 60.00%  (3/5) ########
#########################################
```

In addition, we inform to the user if a program trace did not advance the automaton, i.e., it traverses only the edge that goes to the initial state of the automaton during the verification. Although we cannot observe this situation in this example, it can be viewed in the case study of Section 6.2.1. Other situations, like property violations, can also be viewed in the case studies of Section 6.2.

6.2 Case Studies

To analyze the JPF-SPS performance, we applied it to a set of four Java programs. Two of these programs have clear program traces that can be represented using labeled transition systems [Tre08]: the coffee machine (Section 6.2.1); and, the train door controller [ZBdC$^+$11] (Section 6.2.2. The other two are bigger programs (more than 1000 lines of code) that were used to point the JPF-SPS strengths and weaknesses: the coordinated atomic action (CAA) framework [HM12] and the SWPDC [Ach13, SJ11]. The CAA framework deals with concurrency, and makes use of several threads and loops in its code. The SWPDC is a Java implementation based on a formal specification of an on-board embedded software for a space system developed at INPE.

⇒ The hardware we used to run the case studies is a *MacBook Pro* with a 2.4 GHz Intel Core i5 processor and 8 GB of RAM memory. To run the JPF-SPS, we used the *Eclipse Kepler Integrated Development Environment (IDE)* [Ecl] with *Java version 7*.

6.2.1 Coffee Machine

In the coffee machine example, the user inserts a coin in the machine and can choose one of four options:

(a) coffee and milk;
(b) coffee and sugar;
(c) milk and sugar;
(d) sugar.

The partial code is presented in Listing 6.3. To make JPF to explore all the four options in one execution of the program, we used the "*Verify.random*" method from JPF[Jav].

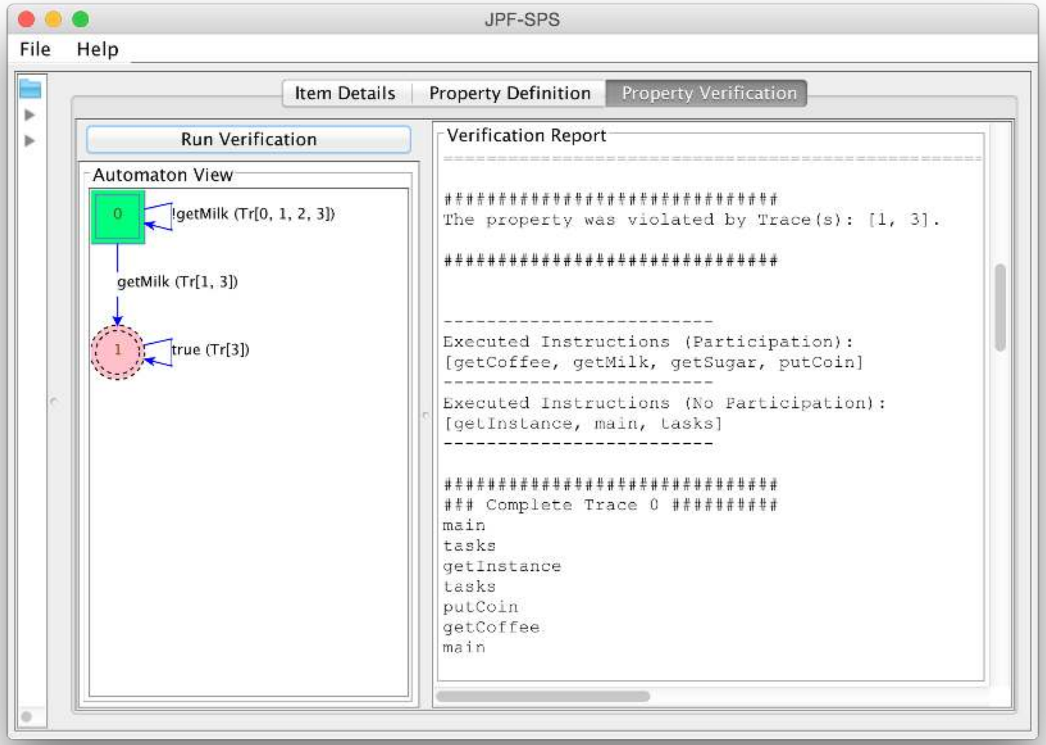
Listing 6.3: *Coffee Machine - Partial Code*

```java
public class CoffeeMachine {
        private static CoffeeMachine cm = null;
        public static void main(String[] args) throws IOException {
                tasks(Verify.random(3));
        }
        private static void tasks(int option) {
                cm = getInstance();
                cm.putCoin();
                switch (option) {
                case 1:
                        cm.getCoffee();
                        cm.getMilk();
                        break;
                case 2:
                        cm.getCoffee();
                        cm.getSugar();
                        break;
                case 3:
                        cm.getMilk();
                        cm.getSugar();
                default:
                        cm.getSugar();
                        break;
                }
        }
        ...
}
```

Suppose that we want to define the property:

Property: `getMilk` cannot be called during the whole execution of the program. This property can be specified using the *absence* pattern for `P = getMilk`, and the *global* scope.

Following the steps that were presented in Section 6.1.1, we can define this property and verify it with JPF-SPS. Figure 6.9 presents the screenshot of the *"Property Verification"* tab in this example. The automaton can be viewed on the left-hand side of the tab, and a partial view of the verification report can be viewed on the right-hand side of the tab.



```
================================================================

################################
The property was violated by Trace(s): [1, 3].

################################


--------------------------
Executed Instructions (Participation):
[getCoffee, getMilk, getSugar, putCoin]
--------------------------
Executed Instructions (No Participation):
[getInstance, main, tasks]
--------------------------

################################
### Complete Trace 0 ##########
main
tasks
getInstance
tasks
putCoin
getCoffee
main
```

Figure 6.9: *Screenshot: Property Verification - Coffee Machine Example*

The verification report (Listing 6.4) presents to the user the property verification verdict, program traces and the automaton edges that were traversed during verification. In this example, the program traces are:

```
(0) putCoin → getSugar;
(1) putCoin → getCoffee → getMilk;
(2) putCoin → getCoffee → getSugar;
(3) putCoin → getMilk → getSugar
```

The property was violated since there are occurrences of `getMilk` in traces 1 and 3, but all the automaton edges were covered. Furthermore, in this example, we also inform to the user the program traces that did not advance the automaton. This case can be observed in traces 0 and 2, which traverse just the edge from node 0 to node 0 (label: `!getMilk`). If we *only* had these program traces, we would not observe the property violation, but the low coverage (1 out of 3 edges) would reveal that the property was not verified properly. This would mean that there may be a problem between the specification and the code. Thus, either the specification or the code should be reviewed.

Listing 6.4: *Coffee Machine example - Partial verification report*

```
JavaPathfinder core system v8.0 (rev 2+) - (C) 2005-2014 United States Government.
All rights reserved.
======================================= system under test
```

```
coffeeMachine.CoffeeMachine.main()
======================================= search started: 8/18/15 10:26 PM
(...)
======================================= search finished: 8/18/15 10:26 PM
###############################
The property was violated by Trace(s): [1, 3].
###############################
Executed Instructions (Participation):
[getCoffee, getMilk, getSugar, putCoin]
-------------------------
Executed Instructions (No Participation):
[getInstance, main, tasks]
-------------------------
(...)
######################
### Trace 0 ##########
putCoin
getSugar

--- Traversed Edges in this Trace: -----
(Node 0, putCoin / !getMilk, Node 0)
(Node 0, getSugar / !getMilk, Node 0)
######################
### Trace 1 ##########
putCoin
getCoffee
getMilk

--- Traversed Edges in this Trace: -----
(Node 0, putCoin / !getMilk, Node 0)
(Node 0, getCoffee / !getMilk, Node 0)
(Node 0, getMilk / getMilk, Node 1)
######################
### Trace 2 ##########
putCoin
getCoffee
getSugar

--- Traversed Edges in this Trace: -----
(Node 0, putCoin / !getMilk, Node 0)
(Node 0, getCoffee / !getMilk, Node 0)
(Node 0, getSugar / !getMilk, Node 0)
######################
### Trace 3 ##########
putCoin
getMilk
getSugar

--- Traversed Edges in this Trace: -----
(Node 0, putCoin / !getMilk, Node 0)
(Node 0, getMilk / getMilk, Node 1)
(Node 1, getSugar / true, Node 1)
########################################
### Traversed Automaton Edges  ##########
(Node 0, !getMilk, Node 0)
(Node 0, getMilk, Node 1)
(Node 1, true, Node 1)
########################################
### Non-Traversed Automaton Edges  ##########

### Edges Coverage 100.00%  (3/3) ########

[WARNING] There is at least one trace that did not advance the automaton:
- Trace: 0
- Trace: 2
```

6.2.2 Train Door Controller

The train door controller code (Listing 6.5) is based on the example presented by Zoppi et al. [ZBdC+11]. It simulates a train door operation when the train departs from a station and arrives at the next station. The door methods are:

`Start`: *the train departs from the station.*
`Alarm`: *the emergency button is pressed.*
`Safe`: *it assures that the train can proceed the trip safely.*
`Stop`: *the train arrives at the next station.*
`Open`: *the doors are opened when the train stops at the station.*
`Close`: *the doors are closed and the train prepares for the next trip.*
`Error`: *it reports an error in the system.*

Listing 6.5: *Train Door Controller - Java version based on the code of Zoppi et al. [ZBdC+11]*

```java
import gov.nasa.jpf.vm.Verify;
public class Controller {
        private static Door door = null;
        public static void main(String[] args) {
                door = new Door();
                scenarioSelection (Verify.random(2));
        }
        public static void scenario1 () {
                door.Start();
                door.Alarm();
                door.Safe();
                door.Stop();
                door.Open();
                door.Close();
        }
        public static void scenario2 () {
                door.Start();
                door.Stop();
                door.Open();
                door.Alarm();
        }
        public static void scenario3 () {
                door.Start();
                door.Alarm();
                door.Stop();
        }
        public static void scenarioSelection (int option) {
                switch (option) {
                case 1:
                        scenario1();
                        System.out.println("---");
                        break;
                case 2:
                        scenario2();
                        System.out.println("---");
                        break;
                default:
                        scenario3();
                        System.out.println("---");
                        break;
                }
        }
}
public class Door {
        private boolean danger, closed, moving;
        public Door() {
                closed = true;
                moving = false;
                danger = false;
        }
        // Controlled operations
        public void Open () {
                if (closed && !moving) {
                        System.out.println("OPEN: Door open; Train not moving");
55                              closed = false;
56                      } else Error();
57              }
58              public void Close () {
59                      if (!closed && !danger) {
60                              System.out.println("CLOSE: Door closed; No danger");
61                              closed = true;
62                      } else Error();
63              }
64              // Monitored events
65              public void Start () {
66                      if (!moving) {
67                              System.out.println("START: Train moving");
68                              moving = true;
69                      }
70                      if (!danger) {
71                              System.out.println("START: Closing doors");
72                              closed = true;
73                      }
74                      else Error();
75              }
76              public void Stop () {
77                      if(moving) {
78                              System.out.println("STOP: Train stopped");
79                              moving = false;
80                      } else Error();
81              }
82              public void Alarm () {
83                      if (!danger) {
84                              System.out.println("ALARM: train in danger, doors opening");
85                              closed = false;
86                              danger = true;
87                      } else Error();
88              }
89              public void Safe () {
90                      if (danger) {
91                              System.out.println("SAFE: Train is safe, no danger");
92                              danger = false;
93                              //closed = true;
94                      } else Error();
95              }
96              public void Error () {
97                      System.out.println("SYSTEM ERROR");
98              }
99      }
```

Suppose that we want to check the following property:

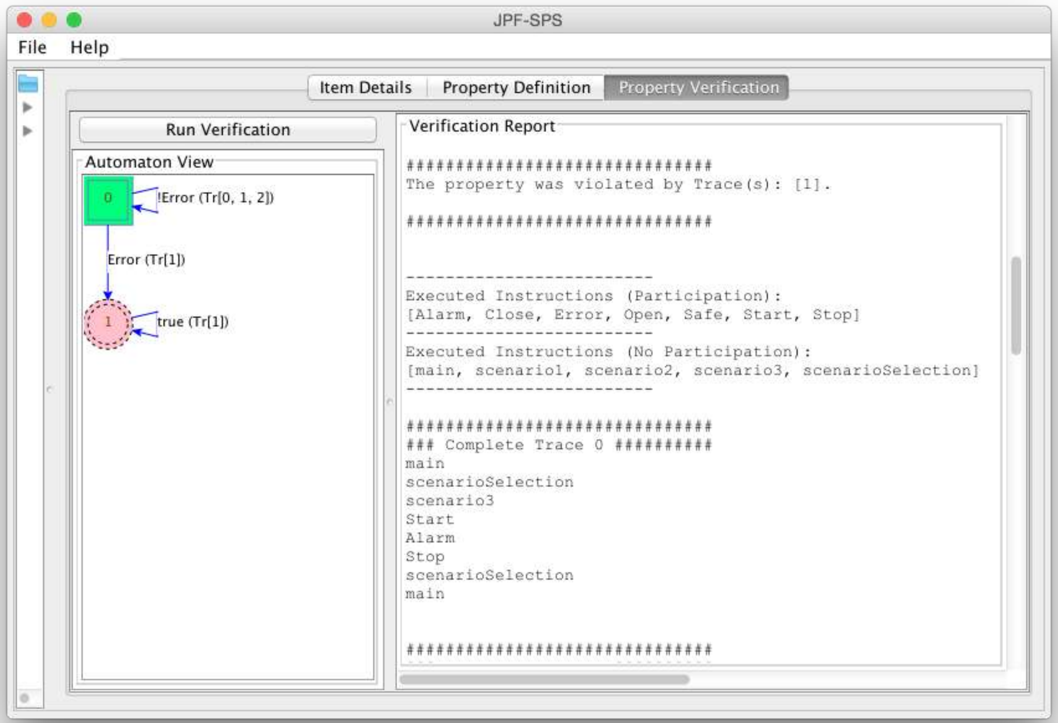
Property 1: `Error` cannot be called during the whole execution of the program. This property can be specified using the *absence* pattern for `P = Error`, and the *global* scope.

This property is similar to the one presented in the paper of Zoppi et al. [ZBdC+11]. And, as we can see in Figure 6.10, the property is violated.

Analyzing the example, we have three different scenarios for the train door operation. These scenarios are verified in the following order:

```
(0) Start → Alarm → Stop
(1) Start → Alarm → Safe →Stop → Open → Close
(2) Start → Stop → Open → Alarm
```

The problem occurs in *scenario 1*. According to the program code, the doors can only be opened if they are currently closed. In scenario 1, `Start` is called, and the doors are closed. Then, `Alarm` is called and the doors are opened. But, after `Safe` is called, the doors are still opened (they were not closed in the `Safe` method). Thus, when `Open` is called, `Error` is called. This program trace can be observed in Listing 6.6. If we want to fix this error, we have to uncomment the instruction at line 93, which will close the doors during the call to the method `Safe`.



```
################################
The property was violated by Trace(s): [1].

################################


-------------------------
Executed Instructions (Participation):
[Alarm, Close, Error, Open, Safe, Start, Stop]
-------------------------
Executed Instructions (No Participation):
[main, scenario1, scenario2, scenario3, scenarioSelection]
-------------------------

################################
### Complete Trace 0 ##########
main
scenarioSelection
scenario3
Start
Alarm
Stop
scenarioSelection
main


################################
```

Figure 6.10: *Screenshot: Property 1 Verification - Train Door Controller Example*

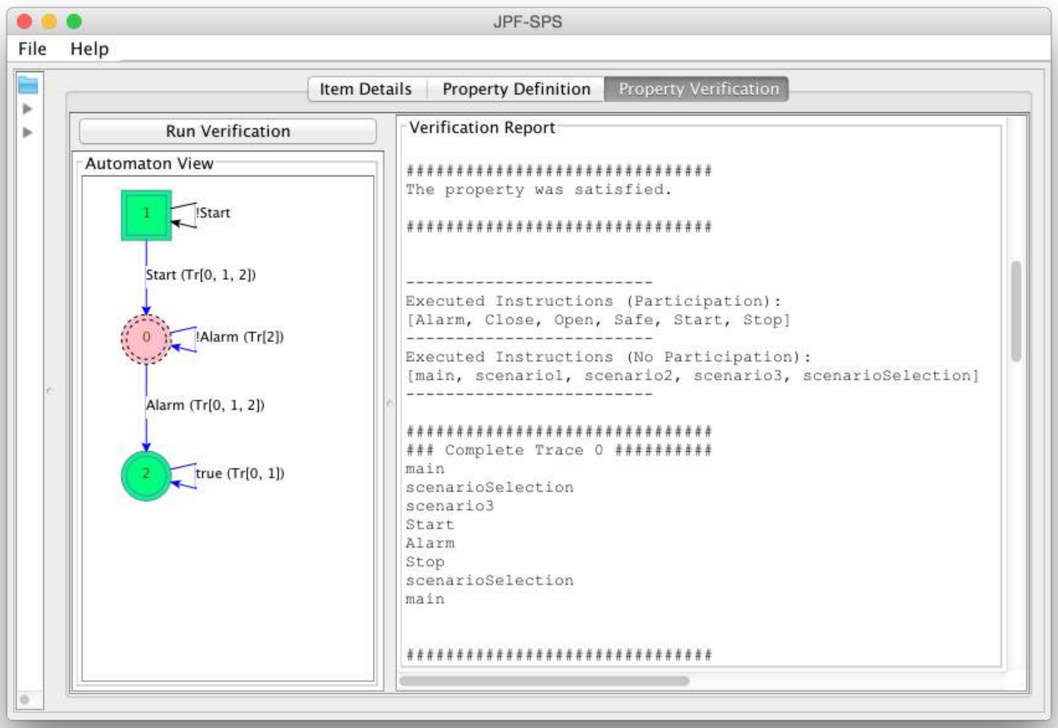
Listing 6.6: *Train Door Controller example - Partial verification report of property 1*

```
JavaPathfinder core system v8.0 (rev 2+) - (C) 2005-2014 United States Government.
All rights reserved.
======================================= system under test
train.Controller.main()
======================================= search started: 8/18/15 11:28 PM
(...)
======================================= search finished: 8/18/15 11:28 PM
################################
The property was violated by Trace(s): [1].
################################
(...)
######################
### Trace 1 ##########
Start
Alarm
Safe
Stop
Open
Error
Close
(...)
```

Now, suppose that we want to check the property:

> **Property 2:** after the method `Start` is called, there is at least one occurrence of `Alarm` *(we want to simulate an emergency after the train starts in all scenarios)*. This property can be defined using the *existence* pattern for `P = Alarm` and the *after* scope for `L = Start`.

In addition, consider that the line 93 of Listing 6.5 is without a comment. As we can see by analyzing the program scenarios, the property is satisfied (Figure 6.11).



```
################################
The property was satisfied.
################################


-------------------------
Executed Instructions (Participation):
[Alarm, Close, Open, Safe, Start, Stop]
-------------------------
Executed Instructions (No Participation):
[main, scenario1, scenario2, scenario3, scenarioSelection]
-------------------------

################################
### Complete Trace 0 ##########
main
scenarioSelection
scenario3
Start
Alarm
Stop
scenarioSelection
main

################################
```

Figure 6.11: *Screenshot: Property 2 Verification - Train Door Controller Example*

The verification report would also state that the property was satisfied after all program traces were analyzed (Listing 6.7).

Listing 6.7: *Train Door Controller example - Partial verification report of property 2*

```
JavaPathfinder core system v8.0 (rev 2+) - (C) 2005-2014 United States Government.
All rights reserved.
======================================= system under test
train.Controller.main()
======================================= search started: 8/18/15 11:52 PM
(...)
======================================= search finished: 8/18/15 11:52 PM
###############################
The property was satisfied.
###############################
(...)
######################
### Trace 0 ##########
Start
Alarm
Stop

--- Traversed Edges in this Trace: -----
(Node 1, Start / Start, Node 0)
(Node 0, Alarm / Alarm, Node 2)
(Node 2, Stop / true, Node 2)
######################
### Trace 1 ##########
Start
Alarm
```

```
Safe
Stop
Open
Close

--- Traversed Edges in this Trace: -----
(Node 1, Start / Start, Node 0)
(Node 0, Alarm / Alarm, Node 2)
(Node 2, Safe / true, Node 2)
(Node 2, Stop / true, Node 2)
(Node 2, Open / true, Node 2)
(Node 2, Close / true, Node 2)
######################
### Trace 2 ##########
Start
Stop
Open
Alarm

--- Traversed Edges in this Trace: -----
(Node 1, Start / Start, Node 0)
(Node 0, Stop / !Alarm, Node 0)
(Node 0, Open / !Alarm, Node 0)
(Node 0, Alarm / Alarm, Node 2)
#########################################
### Traversed Automaton Edges  ##########
(Node 1, Start, Node 0)
(Node 0, Alarm, Node 2)
(Node 2, true, Node 2)
(Node 0, !Alarm, Node 0)
#########################################
### Non-Traversed Automaton Edges  ##########
(Node 1, !Start, Node 1)

### Edges Coverage 80.00%  (4/5) ########
```

According to the verification report (Listing 6.7), 4 out of 5 edges of the automaton were covered, resulting in 80% of coverage (only the edge from node 1 to node 1 (label: $\neg$ `Start`) of the automaton presented in Figure 6.11 was not covered). Since the edge that was not covered is not relevant for the property verification, the generation of test cases may be skipped; otherwise, if the uncovered edge was indeed relevant for test and/or verification purposes, we could use it to generate complementary test cases and reach a 100% coverage of the property automaton.

6.2.3 Coordinated Atomic Actions (CAA) Framework

The framework [Han10, HM12] is based on a specification of the model of coordinated atomic actions (CAA) [Rom01, XRR+95, PdM10]. The CAA model [XRR+95] integrates the concepts of atomic actions [Ran75], ACID transactions [GR92] and concurrent exception handling, aiming at concurrency coordination, communication and error recovery among many system components. Each CAA has a set of participating objects (or participants, for short) that interact with each other in a cooperative manner to accomplish a task.

The CAA framework has approximately 1800 lines of code, and due to the nature of the CAA model, it deals with concurrency among several threads. The threads are used to simulate the CAA model components and their interactions. Figure 6.12 gives an overview of the framework components. We give a brief description of each component next.

The `CoordinatedAtomicAction` class represents one CAA. Each action has one instance of `Coordinator` and `ExceptionTree` classes; and, n instances of `Participant` classes, which are controlled by the `Coordinator` instance.

Each `Participant` has one instance of `Role` and one instance of `Handler` classes. The `Role` instance contains the implementation of the normal behavior of the participant in the action, and the Handler, the implementation of the exceptional behavior of the participant.

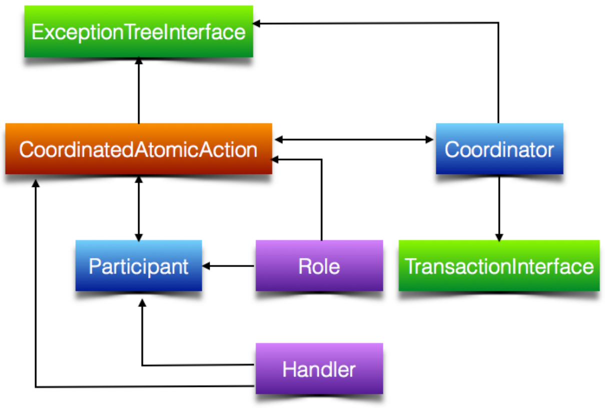


Figure 6.12: *CAA Framework Overview*

Although the CAA framework is bigger than previous examples, and makes use of different data structures to deal with a multithreaded environment ,JPF-SPS was able to handle it: the parsing of the code and properties specification was done in a fast and efficient manner. The restriction is in the verification part, due to the JPF search depth limit: since we disabled the JPF state-matching, we have to set a small limit so JPF can handle all the threads and loops of the CAA framework (otherwise JPF could fall into an infinite loop and overload the hardware memory). The result is that we have a restricted set of properties we can verify until the search depth limit is reached.

Consider a Simulator class that uses the CAA framework. This simulator has two coordinated atomic actions, caa0 and caa1. The first CAA has only one participant (participant0), and the latter has two participants (participant0 and participant1). In the main method, these actions and their components are initialized and executed.

Listing 6.8: *CAA Framework - Simulator Partial Code [Han10]*

```
public class Simulator {
 ...
 public static void main(String args[]) {
  roles0 = new Vector<Role>();
  roles1 = new Vector<Role>();
  handlers0 = new Vector<Handler>();
  handlers1 = new Vector<Handler>();
  participants0 = new Vector<Participant>();
  participants1 = new Vector<Participant>();
  transactions0 = new Vector<TransactionInterface>();
  transactions1 = new Vector<TransactionInterface>();
  exceptions = new Vector<Exception>();
  nestedActions = new Vector <CoordinatedAtomicAction>();
  caaVector = new Vector<CoordinatedAtomicAction>();

  //exception resolution tree
  excTree = new ExceptionTree();

  //CAA-0 and its transactions vector
  caa0 = new CoordinatedAtomicAction(0);
  t0 = new Transaction();
  transactions0.add(t0);
  //CAA-1 and its transactions vector
  caa1 = new CoordinatedAtomicAction(1);
  t1 = new Transaction();
  transactions1.add(t1);

  //adding CAAs to the vector 'caaVector'
  caaVector.add(caa0);
  caaVector.add(caa1);

  //Participant-0 and its roles and handlers
  p0 = new Participant0(caaVector, 0);
  //adding account number 1 at the database
```

```java
  r0 = new Participant0Role0(p0, caa0, 1);
  roles0.add(r0);
  p0.setRoles(roles0);
  h0 = new Participant0Handler0(p0, caa0);
  handlers0.add(h0);
  p0.setHandlers(handlers0);
  //adding Participant-0 in the participant's vector of the CAA-0
  participants0.add(p0);

  //Participant-1 and its roles and handlers
  p1 = new Participant1(caaVector, 1);
  r1 = new Participant1Role0(p1, caa0);
  r2 = new Participant1Role1(p1, caa1);
  roles1.add(r1);
  roles1.add(r2);
  p1.setRoles(roles1);
  h1 = new Participant1Handler0(p1, caa0);
  handlers1.add(h1);
  p1.setHandlers(handlers1);

  //adding Participant-1 in the participant's vector of the CAA-0
  participants0.add(p1);
  //adding Participant-1 in the participant's vector of the CAA-1
  participants1.add(p1);
  //adding the 'InvalidAccountNumberException' to the exceptions' vector of
  //CAAs.
  InvalidAccountNumberException exc = new InvalidAccountNumberException();
  exceptions.add(exc);

  //setting CAA-0 variables
  caa0.setTrans(transactions0);
  caa0.setParticipants(participants0);
  nestedActions.add(caa1);
  caa0.setNestedActions(nestedActions);
  caa0.setExceptions(exceptions);
  caa0.setExceptionTree(excTree);
  //CAA-0 coordinator
  coord0 = new Coordinator(caa0);
  caa0.setCoordinator(coord0);

  //setting CAA-1 variables
  caa1.setTrans(transactions1);
  caa1.setParticipants(participants1);
  caa1.setNestedActions(new Vector<CoordinatedAtomicAction>());
  caa1.setExceptions(exceptions);
  caa1.setExceptionTree(excTree);
  caa1.setParentCaa(caa0);
  //CAA-1 coordinator
  coord1 = new Coordinator(caa1);
  coord1.setfSignal(exceptions);
  caa1.setCoordinator(coord1);

  //starting participants execution
  for (Participant partic : participants0) {
    partic.start();
  }
  //starting coordinators execution
  caa0.getCoordinator().start();
  caa1.getCoordinator().start();
 }
}
```

Moreover, consider the `Participant` class methods (Listing 6.9).

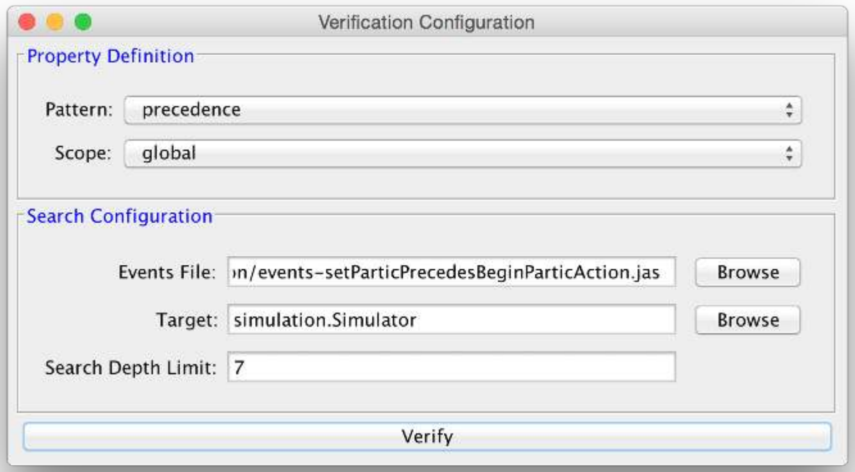
Listing 6.9: *CAA Framework - Participant Partial Code [Han10]*

```java
public class Participant extends Thread {
  ...
    //this method requests the participant entrance in a CAA.
    public void entryRequest(int caaIndex) {...}
    //this method requests the normal ending of a participant in a CAA.
    public void normalEndRequest(int caaIndex) {...}
    //this method requests the exceptional ending of a participant in a CAA.
    public void exceptionalEndRequest(int caaIndex) {...}
    //this method requests the abort ending of a participant in a CAA.
    public void abortEndRequest(int caaIndex) {...}
    //this method does all the participant acting in the CAA with
    //index caaIndex.
    public void beginParticipantAction(int caaIndex) {...}
    //this method does the normal or exceptional ending of a handler.
    private void checkParticipantHandlerEnding(int caaIndex, int handlerVindex) {...}
    //this method does the normal or exceptional ending of a role.
    private void checkParticipantRoleEnding(int caaIndex, int roleVindex) {...}
    //Getters and Setters
    ...
}
```

In spite of the search limit, we are not restricted to the verification of properties regarding methods that are called in the `main` method of the `Simulator` class (Listing 6.8). We can also verify properties regarding methods that belong to other classes whose instances are used in the `main` method. Focusing on the methods concerning to the `Participant` instances, consider the following property:

Property: we want to assure that a call to `setParticipants` always precedes a call to `beginParticipantAction`. This guarantees that the coordinated atomic action has been configured before the participant actions start.

The property can be defined using the *precedence* pattern in the *global* scope. In the pattern specification, we have that `T = setParticipants` and `P = beginParticipantAction`. Setting the verification with this property, and putting the search limit as 7, we can run the verification (Figure 6.13).


Verification Configuration

Property Definition

Pattern: precedence

Scope: global

Search Configuration

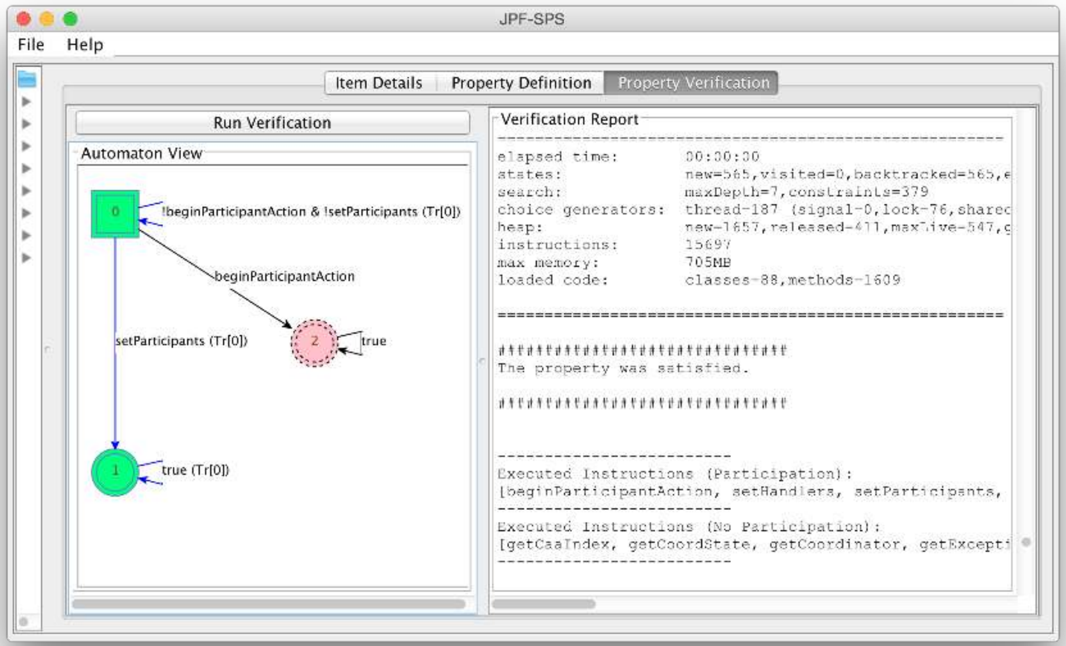
Events File: ›n/events-setParticPrecedesBeginParticAction.jas

Target: simulation.Simulator

Search Depth Limit: 7

Figure 6.13: *Screenshot: Verification Configuration - CAA Framework Example*

The result is that the property is satisfied by the simulator. Figure 6.14 presents the property automaton view and part of the verification report. The automaton confirms that the property is satisfied, since it does not reach the rejecting state 2. A better view of the verification report is presented in Listing 6.10. From this listing, we can observe that the verification report of this example is different from the reports of previous examples. This fact occurs due to the interruption of the verification process when the search limit is reached. As we have discussed in Section 5.2.2, the listener (Listing 5.1) prints the program trace and property automaton coverage until the moment of the interruption. The property verdict is reached by analyzing the automaton coverage. Since the automaton is traversed according to the instruction that is executed by the model checker, it would still give the right coverage and the right verdict until the program execution is interrupted.



```
---------------------------------------------------------
elapsed time:        00:00:00
states:              new=565,visited=0,backtracked=565,e
search:              maxDepth=7,constraints=379
choice generators:   thread-187 (signal-0,lock-76,sharec
heap:                new-1657,released-411,maxLive-547,g
instructions:        15697
max memory:          705MB
loaded code:         classes-88,methods-1609

=========================================================

################################
The property was satisfied.

################################


--------------------------
Executed Instructions (Participation):
[beginParticipantAction, setHandlers, setParticipants,
--------------------------
Executed Instructions (No Participation):
[getCaaIndex, getCoordState, getCoordinator, getExcepti
--------------------------
```

Figure 6.14: *Screenshot: Property Verification - CAA Framework Example*

Therefore, the property verdict can be guaranteed only in the executed portion of the program, i.e., the part of the program that was executed until the interruption.

Listing 6.10: *CAA Framework example - Partial verification report*

```
JavaPathfinder core system v8.0 (rev 2+) - (C) 2005-2014 United States Government.
All rights reserved.
======================================= system under test
simulation.Simulator.main()
======================================= search started: 8/20/15 12:46 AM
(...)
====================================================== search constraint
depth limit reached: 7
(...)
======================================= search finished: 8/20/15 12:46 AM
###############################
The property was satisfied.
###############################
-------------------------
Executed Instructions (Participation):
[beginParticipantAction, setHandlers, setParticipants, setRoles]
-------------------------
Executed Instructions (No Participation):
[getCaaIndex, getCoordState, getCoordinator, getExceptionTree, getIndex,
   getNestedActions, getParentCaa, getParticipants, getPropagatedExceptions, getTrans,
    main, run, setCoordinator, setExceptionTree, setExceptions, setNestedActions,
   setParentCaa, setTrans, setfSignal]
-------------------------
(...)
######################
### Trace 0 ##########
setRoles
setHandlers
setRoles
```

```
setHandlers
setParticipants
setParticipants
beginParticipantAction
beginParticipantAction
beginParticipantAction
beginParticipantAction
beginParticipantAction
beginParticipantAction
beginParticipantAction
beginParticipantAction
beginParticipantAction
beginParticipantAction
beginParticipantAction
(...)
#########################################
### Traversed Automaton Edges  ##########
(Node 0, !beginParticipantAction & !setParticipants, Node 0)
(Node 0, setParticipants, Node 1)
(Node 1, true, Node 1)
#########################################
### Non-Traversed Automaton Edges  ##########
(Node 0, beginParticipantAction, Node 2)
(Node 2, true, Node 2)

### Edges Coverage 60.00%  (3/5) ########
```

6.2.4 SWPDC

SWPDC [Ach13] is a Java implementation of a real space software product developed for an INPE's research project [SJ11]. It has approximately 1400 lines of code. Figure 6.15 gives an overview of the implemented SWPDC components and their interactions.

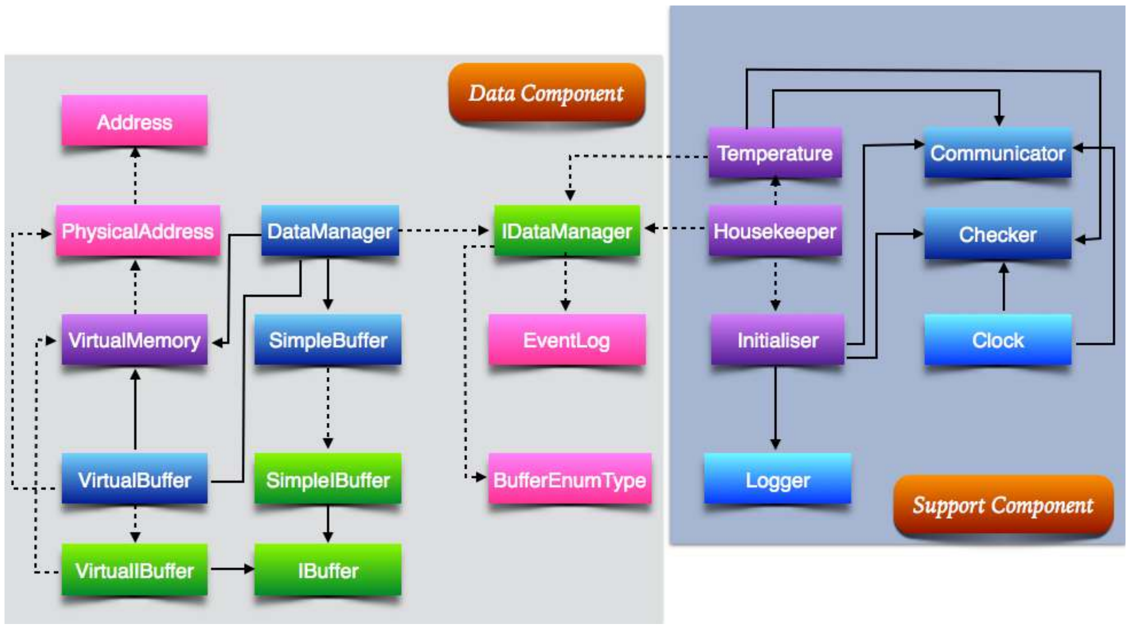


Figure 6.15: *SWPDC Components Overview*

The *data component* is responsible for the whole data management, from its storage to its formatting. On the other hand, the *support component* is responsible for initializing the sensor procedures and the sensor data acquisition. To implement the communication between these two components, text files were used to simulate the data exchange.

JPF-SPS could handle the SWPDC program similarly to the way it handled the CAA framework (Section 6.2.3). The parsing of the code and properties specification was done in a fast and efficient manner. The restriction is also in the verification part: JPF cannot deal with IO libraries that were used in the implementation of SWPDC. Hence, the SWPDC `main` method was adapted to try to make JPF skip the methods that use these libraries. The result is that the program could not be fully verified, and the properties were restricted to the events that do not belong to program execution traces with IO methods that are not supported by JPF.

Reproducing the example presented by Achutti [Ach13], we use the proposed SWPDC class with an adapted main to run with JPF-SPS. This class code is shown in Listing 6.11.

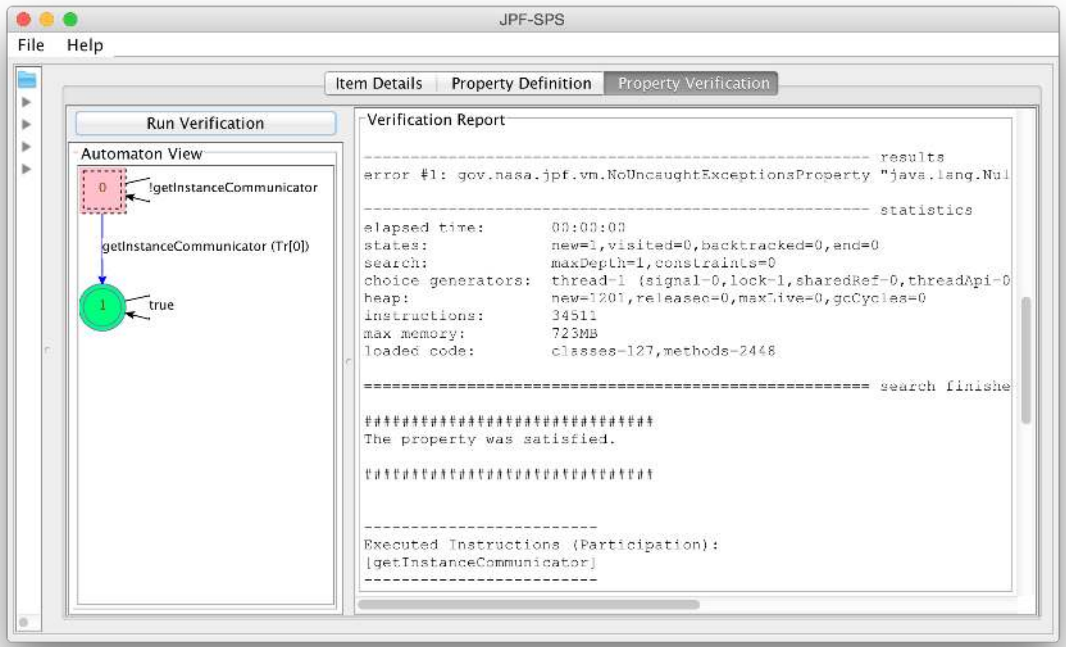
Listing 6.11: *SWPDC - Class with an adapted main method [Ach13]*

```java
public class MainForSupportVerification {
    static Communicator communicator;
    static Checker checker;
    static Initialiser initialiser;

    public static void main(String[] args){
        communicator = Communicator.getInstanceCommunicator();
        checker = Checker.getInstanceChecker();
        initialiser = new Initialiser();
        initialiser.begin();
    }
}
```

Considering the set of properties that were implemented with SWPDC, we have chosen to specify one of the properties related to the support component. The property checks the occurrence of a call to `getInstanceCommunicator`, and can be defined as follows:

> **Property:** there is at least one call to `getInstanceCommunicator` during the whole program execution. This property can be specified using the *existence* pattern in the *global* scope.

Defining this property is an easy task with JPF-SPS. But we still observed some restrictions when verifying this property against the adapted SWPDC class (Listing 6.11). Figure 6.16 presents the property automaton and the partial verification report.



```
------------------------------------------------------- results
error #1: gov.nasa.jpf.vm.NoUncaughtExceptionsProperty "java.lang.Nul

------------------------------------------------------- statistics
elapsed time:       00:00:00
states:             new=1,visited=0,backtracked=0,end=0
search:             maxDepth=1,constraints=0
choice generators:  thread=1 (signal=0,lock=1,sharedRef=0,threadApi=0
heap:               new=1201,released=0,maxLive=0,gcCycles=0
instructions:       34511
max memory:         723MB
loaded code:        classes=127,methods=2448

======================================================= search finishe

##############################
The property was satisfied.

##############################


-------------------------
Executed Instructions (Participation):
[getInstanceCommunicator]
-------------------------
```

Figure 6.16: *Screenshot: Property Verification - SWPDC Example*

The partial verification report concerning to this property is presented in Listing 6.12.

Listing 6.12: *SWPDC example - Partial verification report*

```
JavaPathfinder core system v8.0 (rev 2+) - (C) 2005-2014 United States Government.
All rights reserved.
======================================= system under test
swpdc.MainForSupportVerification.main()
======================================= search started: 8/20/15 1:05 AM
(...)
====================================================== error 1
gov.nasa.jpf.vm.NoUncaughtExceptionsProperty
(...)
======================================= search finished: 8/20/15 1:05 AM
###############################
The property was satisfied.
###############################
-------------------------
Executed Instructions (Participation):
[getInstanceCommunicator]
-------------------------
Executed Instructions (No Participation):
[createFile, getDataTime, main]
-------------------------
###############################
### Complete Trace 0 ##########
main
getInstanceCommunicator
createFile
getDataTime
createFile
######################
### Trace 0 ##########
getInstanceCommunicator

--- Traversed Edges in this Trace: -----
(Node 0, getInstanceCommunicator / getInstanceCommunicator, Node 1)
#########################################
### Traversed Automaton Edges  ##########
(Node 0, getInstanceCommunicator, Node 1)
#########################################
### Non-Traversed Automaton Edges  ##########
(Node 0, !getInstanceCommunicator, Node 0)
(Node 1, true, Node 1)

### Edges Coverage 33.33%  (1/3) ########
```

Similarly to the previous example (Section 6.2.3), the result is that the property is satisfied by the program. This result is confirmed by the property automaton (Figure 6.16), since it only reaches the accepting state 1. Nevertheless, the verification report that is presented in Listing 6.12 presents an error. Recall that the *NoUncaughtExceptionProperty* is a native property of JPF, and it is violated when an exception is not handled correctly by the IUT. In the SWPDC example, the error occurs when an instruction concerning to file manipulation is called and it does not handle the exception properly according to JPF (observe that in Listing 6.12 there is a call to a method `createFile`). Therefore, as in the CAA framework example, the SWPDC verification is also interrupted. Although the interruption causes are different, the result is similar. The listener (Section 5.2.2, Listing 5.1) prints the program trace and property automaton coverage until the moment of the interruption. The property verdict is reached by analyzing the automaton coverage. Note that this analysis is only valid until the point of the interruption of program execution, hence, the verdict analysis and coverage data is only valid for the portion of the program that was indeed executed during the model checking.

Even though our property was satisfied, the error that is reported during the verification should not be ignored. Therefore, the most appropriate solution in case of a native JPF property violation is the review of the program code to fix the error. If the error was caused by a method that uses libraries that are not supported by JPF, we could either skip the call to this method in the verification or change the implementation of this method to avoid the use of unsupported libraries.

6.3 Conclusion

In this chapter, we presented the JPF-SPS, the tool we implemented to integrate testable properties classification and Java model checking. After we give an overview of the tool and its features, we evaluated it through a set of Java examples. We could observe that the identification of program elements (e.g. classes, methods) and the definition of properties using the proposed testable properties classification (Chapter 3) can be done in a fast and efficient manner regardless of the size of the example program. The main restrictions concern to the back-end of the tool, the verification engine. Although JPF is a powerful tool, it does not offer full support for all Java libraries. For instance, it cannot deal properly with *IO libraries* and *Java Swing*. In addition, to adapt JPF to our tool trace analysis, we had to disable the JPF state-matching. Consequently, when JPF-SPS receives a program with recursion or loop structures, we have to put a limit to the JPF search for property violation, thus constraining the portion of the program that is effectively verified. Due to these JPF restrictions, the verification may be interrupted, giving to the users only the results concerning to the executed portion of the program. This could be observed in the cases studies of Section 6.2.3 and 6.2.4. In spite of the verification engine restrictions, we can conclude that JPF-SPS does what we wanted it to do: (a) it detects the program elements that can be used to define a property; (b) it provides an intuitive interface to define a testable property; (c) it provides a direct link with a verification engine to verify the defined property against the Java program. *Having these features, JPF-SPS simplifies the use of formal verification in such a way that even users that are not familiar with formal methods can apply formal method in their Java programs.*

Concerning to JPF-SPS additional contributions, and its similarities and differences to other existing tools, we now discuss some related work. Balasubramanian et al. [BPN$^+$11] defined JPF properties using contracts or automata, which are generated using the SPS and then linked to Java code and the JPF using Matlab/Simulink [Mat]. *Although we use automata to represent SPS combinations of pattern and scope, we make a direct relation between the definition of testable properties with the SPS and the property verification using a JPF listener.* The JPF-LTL [JL] extends JPF with LTL verification. *While JPF-LTL is more general, our work focuses on ease of use, via patterns and a user-friendly interface. In addition, we provide property coverage information that can be used to guide a reduced test suite generation.* Regarding property definition, Mondragon et al. [MGR04] developed the tool Prospec. This tool provides a user-friendly interface to assist the definition of properties using SPS, but no automaton is generated. *In contrast, we provide the program events that can be used to specify a property, and the corresponding finite-automaton of the property, which was specified with SPS; and also, we provide a direct link with JPF, allowing the verification of the property.* Belinfante [Bel10] developed the tool JTorX, which checks whether a test purpose specification holds in a Java program implementation using the *ioco* relation. The test purposes are represented in LTS and the underlying theory of *ioco* relation can be found in [Tre08]. Jard and Jéron [JJ05] developed the tool TGV, which generates test cases from test purposes represented using LTS. *In our work, we proposed the use of test purpose as input to a model checker. The results provided by the verification process can then be used to guide test cases generation.*

The next part of this thesis (Part IV) presents the conclusion of the research. The main contributions and achieved results are discussed, as well as future work.

IV— Final Considerations

7 — Concluding Remarks

This chapter concludes the thesis. Here, we present a summary of the accomplished activities and scientific contributions regarding this research. We also discuss the achieved results and future work.

7.1 Research Activities

The research activities were planned having as main goal the accomplishment of this research objectives (Section 1.2). Before we proceed with the summary of these activities, we first recall the objectives of this work that we presented previously:

- **Objective 1.** Development of a methodology to convert properties that are derived from test purposes and specified in a formal language to a formal representation that could be used in a Java formal verifier. This objective can be divided in two parts:
  (a) ***Part I (Theoretical Work).*** This part involves the study of existing approaches for test purposes specification, and the definition of the universe of properties we should handle in our research. To this intent of defining a universe of properties, we must establish a connection between test purposes specification and properties testability. Then, a systematic way to specify a property, and then convert it into a formal representation that can be read by a model checker, must be defined.
  (b) ***Part II (Practical Work).*** This part connects the property specification and formal representation with model checking. A Java model checker (in our case, the Java PathFinder (JPF) [Jav]) must be adapted to read the property formal representation, and verify the property satisfiability during program execution. This part is responsible for providing the mechanisms we need to implement our own tool that covers property specification and formal verification.
- **Objective 2 *(Practical Work).*** Implementation of a tool that will make this feature (simplification of property specification and verification of the defined property with a model checker) available to software developers. This objective is part of the practical work of this research.
- **Objective 3 *(Practical Work).*** Evaluation of the tool through its application in a set of case studies in order to assess the validation of the developed methodology. The approach for this evaluation is not statistical, but it is focused in the analysis of the advantages and restrictions of our tool application in practice. This objective is part of the practical work of this research.

7.1.1 Summary of Accomplished Research Activities

To achieve the objectives of this work, a set of research activities (RA) were planned and accomplished during the development of this research. A summary of these activities and the corresponding results are presented next.

RA1: Preliminary studies (Chapter 2). To start the development of our work, we studied basic concepts regarding property specification and formal verification.

- In the *study of property specification*, we analyzed formal specification languages that are used for this purpose (e.g. linear temporal logic (LTL)[Pnu77], regular expressions [Mit03]), and the specification pattern system (SPS) [DAC99, Pat] whose aim is to assist practitioners in specifying properties through a set of predefined patterns. In addition, we studied existing tools for property specification (e.g. Prospec [MGR04], Spider [KC05a]). After this study, we decided to use LTL, regular expressions and SPS in the development of our work. The LTL is used in the theoretical part or our work, which comprises the analysis of properties testability [NGH93] and correctness of specification; regular expressions are used in the conversion of properties specification into a formal representation that is readable by a model chacker; and SPS are used in the whole process.
- In the *study of formal verification*, we reviewed the model checking process and some of the existing model checkers for Java programs (e.g. Bandera [Ban], Spin [SPI], Java PathFinder (JPF) [Jav]). We concluded that JPF would be the best choice to use in our approach due to the possibility of defining a property specification in Java, the same programming language of the IUT. Furthermore, it can be extended with not only additional properties, but also with listeners that can monitor the program execution during the verification process.

RA2: Study of the connection between testable properties and SPS (Part II). By connecting the concepts of properties testability [NGH93, FFJ$^+$10] to the SPS through the analysis of linear temporal properties, we could establish a classification of testable properties. This classification is composed by a set of SPS formulas that can derive testable properties (i.e., properties that give a success/fail verdict). The properties that are specified using this classification can then be verified with a model checker, which in turn will be able to provide a success/fail verdict for the property (i.e. the model checker will not provide the unknown verdict when verifying a testable property).

RA3: Analysis of the correctness of SPS formulas from the classification of test purposes based on the testability concept (Chapter 4, Appendix C). The study, which confirmed the correctness and validation conditions for SPS formulas through the analysis of LTL, certified that the classification of testable properties can be used not only for properties in the global scope, but it can also be extended and used for properties in local scopes (Chapter 4). Moreover, we did a similar study covering the Prospec [MGR04] formulas, which are slightly different from the SPS original formulas, achieving the same result (Appendix C). Thus, we assured that a broader range of SPS formulas can be used to derive testable properties.

RA4: Conversion of properties specification into a formal representation for model checking (Chapter 5, Appendix D). In RA2, we established the set of SPS patterns and scopes that can be used to specify testable properties. For each combination of these SPS patterns and scopes, we use the corresponding SPS regular expression to generate a minimal deterministic automaton (DFA) that can be inputted into a model checker (in our case, the DFA is used in JPF). Hence, we created a library of automata that represent testable property patterns. Then, to verify a property, the model checker must be able to read the corresponding automaton, and traverse it during program execution to monitor the verification and collect program and property coverage data.

RA5: Implementation of a listener for JPF (Chapter 5). Recalling that in RA1 we decided to use the model checker JPF in our work, we needed to adapt JPF so it could be able to use the automata library we developed in RA4. Thus, to do this adaptation, we implemented a general JPF listener that reads one property automaton at a time, and traverses it to reach a pass/fail verdict. Setting this listener in JPF makes the model checker able to verify the properties automata.

RA6: Evaluation of the correctness of the automata library and JPF listener (Chapter 5). The automata representation was generated by the JFLAP tool [Rod06], which has strong recommendations for use in theoretical courses [JFL], and whose correctness has been proved in a list of publications [Rod06, JFL]. Therefore, we assume that the automata library that was generated by JFLAP is correct. The JPF listener implementation is straightforward, and is correct by construct.

RA7: Implementation of a tool to simplify the property specification and formal verification (Chapter 6). We implemented a tool, JPF-SPS, which integrates testable properties classification and Java model checking. Properties are defined using the SPS formulas from the classification of testable properties; to simplify the task, we provide to the user a friendly interface to define the property with program events (methods), which are identified during the parsing of the Java program. Using the provided information, the tool JPF-SPS automatically selects the automaton that represents the property from its library and replace the automaton default labels by the selected program events. The property automaton is read by a listener and the property is checked by JPF. A verification report with the property pass/fail verdict, program traces and the coverage of the traversed edges of the automaton is presented to the user. Using this information, a test suite for the portion of the property that was not covered can be generated. Therefore, using this approach, we can check a program against a property specification, and generate a reduced test suite.

RA8: Case studies (Chapter 6). The tool, JPF-SPS, was applied to a set of Java programs. The set of examples is composed by different types of Java systems, since we wanted to assess the behavior or JPF-SPS and give a clear step-by-step guide of using it to define and verify properties against different Java programs.

As we can observe in Table 7.1, the set of accomplished activities of this research allow us to conclude that the main objectives of our work (Sections 1.2 and 7.1) were achieved: we provided a simplified approach to specify properties and verify them against their corresponding Java program. The application of our approach relies on the use of JPF-SPS, a tool that integrates the classification of testable properties to generate the minimal DFA of a property, and the JPF, the model checker that reads and traverses the corresponding property automaton while it performs the program verification.

Table 7.1: *Research Objectives and Accomplished Activities*

Objective	**Research Activities (RA)**
Objective 1(a) - Properties formal representation (theory)	RA1, RA2, RA3, RA4
Objective 1(b) - Properties formal representation (practice)	RA1, RA4, RA5, RA6
Objective 2 - Tool implementation (JPF-SPS)	RA7
Objective 3 - Cases studies for tool evaluation	RA8

7.2 Scientific Contributions

The main contributions of this research are: the definition of a classification of testable properties (Chapter 3), the formal demonstration of the correctness of SPS formulas through the analysis of LTL semantics and validation conditions (Chapter 4), and the development of a tool for property definition and verification, which we call as JPF-SPS (Chapter 6). Next, we discuss these contributions and the achieved results.

Classification of Testable Properties. The classification of testable properties is composed by a set of SPS formulas that can derive testable properties (i.e., properties that give a success/fail verdict). In this set, we have the patterns of *universality*, *absence*, *precedence*, *existence* and *response* in their canonical formulas (global scope) and in their formulas for local scopes (*before R*, *after L*, *between L and R*, and *after L until R*). We could establish this classification by connecting the concepts of properties testability (Section 3.1) to the SPS (Section 2.4) through the analysis of linear temporal properties.

⇒ *Main Contribution:* Using this classification, we simplify the process of specifying testable properties in the context of programs with finite execution. The specified properties can then be verified with a model checker, which in turn will be able to provide a success/fail verdict for the property (i.e. the model checker will not provide the unknown verdict when verifying a testable property).

Correctness and Validation Conditions for SPS Formulas. When establishing a classification of testable properties, we observed that the connection of SPS patterns in the global scope with the testability concept can be reached directly. But for the local scopes (*before R*, *after L*, *between L and R*, and *after L until R*), a deeper analysis was required to check whether this result could be extended to those formulas. This analysis was done using the definitions of LTL syntax and semantics (Sections B.2 and B.3). The result is that the SPS patterns (*universality*, *absence*, *precedence*, *existence* and *response*) are able to derive testable properties in all SPS scopes.

⇒ *Main Contribution:* The study, which confirmed the correctness and validation conditions for SPS formulas, certified that the classification of testable properties can be used not only for properties in the global scope, but it can also be extended and used for properties in local scopes. Moreover, we did a similar study covering the Prospec [MGR04] formulas, which are slightly different from the SPS original formulas (Appendix C), achieving the same result. Thus, we assured that a broader range of SPS formulas can be used to derive testable properties. In addition, a formal validation of the SPS formulas concerning to the LTL representation is given.

Tool for Property Definition and Verification (JPF-SPS). After defining a classification of testable properties, and validating the correctness of the SPS formulas from the classification, we wanted to use this information to achieve our main objective of simplifying the application of model checking in practice. Therefore, we defined test purposes as testable properties to be verified by the model checker Java PathFinder (JPF) [Jav]. Properties are defined using the SPS formulas from the classification of testable properties; to simplify the task, we provide to the user a friendly interface to define the property with program events (methods). Using the provided information, the tool JPF-SPS automatically selects the automaton that represents the property from its library and replace the automaton default labels by the selected program events. The property automaton is read by a listener and the property is checked by JPF. A verification report with the property pass/fail verdict, program traces and the coverage of the traversed edges of the automaton is presented to the user. Using this information, a test suite for the portion of the property that was not covered can be generated. Therefore, using this approach, we can check a program against a property specification, and generate a reduced test suite. To show how our approach can be used in practice, we applied our tool to a set of examples.

⇒ *Main Contribution:* The implemented tool, JPF-SPS, provides a more intuitive manner of defining a property through its front-end, a user-friendly interface. Using the interface, users can collect information regarding program events and define testable properties using the SPS. The back-end of the tool is the model checker JPF. By integrating JPF with a user-friendly interface for property definition, we can perform program verification in a simpler and faster manner. In addition, JPF-SPS provides information concerning to the generated property automaton and program verification coverage, which can be used to create complementary test cases.

The main results and scientific contributions of this research have already been published. A list of these publications is presented in Appendix A. As we can observe, the set of contributions of this research allow us to conclude that the main objective of our work was achieved: we provided a simplified approach to specify properties and verify them against their corresponding Java program. The application of our approach relies on the use of JPF-SPS, a tool that integrates the classification of testable properties to generate the minimal DFA of a property, and the JPF, the model checker that reads and traverses the corresponding property automaton while it performs the program verification.

7.2.1 Research Limitations

Regarding the theoretical part of our research, which comprises the study of testable properties and how they can be linked to the SPS, the main limitation is the context of finite execution programs. Even though the theoretical results can be usually applied for programs with either finite or infinite execution, we have chosen to work with the finite execution programs to include the SPS response pattern in the classification of testable properties. Furthermore, working in this context had influenced our decision to work with regular expressions in the practical part of our work. Since regular expressions cannot deal with programs with infinite execution, we kept our whole research in the context of finite execution programs.

Another decision concerning to the theoretical part of our research is the no inclusion of SPS repetition patterns (e.g. response chain, precedence chain). Although the intuition is that they will lead to the specification of testable properties, we would need to deal with the problem of repetition. Since the repetition could be infinite, we would have to either deal with infinite execution programs, or study a way to validate these patterns by constraining the repetition to a finite context.

The practical part of our research, which covers the JPF-SPS implementation, has as main limitation the use of single events with no parameters to define testable properties with SPS. Having this restriction simplified the process of building JPF-SPS, since we had to deal only with method calls during the verification process. The inclusion of program states is a complex task, which would demand more time to develop a theoretical background to support the implementation of JPF-SPS. Combinations of events and the inclusion of events with parameters would also be complex and time-consuming, since a study regarding the effects of parameters inclusion and logical combinations of the events in the definition of testable properties should be carried out. Other restrictions of JPF-SPS are inherited from JPF, as discussed in Section 6.3.

To overcome these research limitations, we suggest a set of activities as future work. New research directions could also be explored, having our research contribution as a starting point. These suggestions are presented in Section 7.3.

7.3 Future Work and Directions

Future work regarding this Doctorate research comprises the extension of properties testability study, and the addition of new features to JPF-SPS. Below, we suggest activities concerning to each of these topics.

Extension of properties testability study:

- Check the feasibility of adding other SPS patterns (e.g. precedence chain, response chain) and Prospec patterns (e.g. strict precedence);
- Check whether the results can be extended to the context of infinite execution programs;
- Study the cost of extending the specification of properties to program states, conditions, and events with parameters; and, check whether the properties keep their testability with this extension.

Improve and add new features to JPF-SPS:

- Add other testable property patterns, according to the previously suggested study;
- Provide to the user sets of test cases that can be derived from the program verification data (e.g. program execution traces, property automaton coverage);
- Further investigate the JPF-SPS scalability and performance, and make improvements according to the obtained results.

Besides those suggested activities, which intend to continue and improve this research, we consider that other directions can be explored using the achieved results of our work as a starting point:

- **Connect the front-end of the developed tool to other verification or testing tools:** study the possibility of replacing JPF with other verification, or even testing tools. By using other tools as JPF-SPS back-end, it may be possible to extend the application of JPF-SPS to other programs and areas that are not covered or supported by JPF.
- **Educational purposes:** Due to its user-friendly interface, and its direct integration with the JPF model checker, JPF-SPS can be a useful tool when teaching students the topic of formal verification. An empirical study is also suggested to assess the easy of use of JPF-SPS.
- **Runtime verification:** Since in JPF-SPS we can monitor the verification using a JPF listener, we could explore the use of this monitoring for a single execution of the program. This would imply a study of the application of the model checker JPF in runtime verification, and the possible inclusion of this feature in JPF-SPS.

V— Appendix and References

A — Publications

In this appendix, we present a list of published papers from Simone Hanazumi. The papers were written during her Doctorate, and describe either results from the study of property specification, testing requirements, formal verification and the application of JPF, or scientific contributions that are directly related to her Doctorate research. A list of papers that are currently under review is also presented.

A.1 Published Papers

This section presents the list of published papers according to the chronological order of publication. The papers are organized by year.

A.1.1 Year: 2012

Title: *"Coordinating Exceptions of Java Systems: Implementation and Formal Verification"*
Authors: Simone Hanazumi and Ana C. V. de Melo
Published in: Proceedings of the 8th International Conference on the Quality of Information and Communications Technology (QUATIC 2012)
Date: September 2012

Abstract:
The exceptional behavior of software has become an important issue in software development since software may collapse if exception-handling is not implemented accordingly. Aiming at this problem, the Coordinated Atomic Action (CAA) model was proposed: it guides users to treat exceptions in a well-organized way, maintaining the whole system stable. However, deriving a system implementation from a CAA specification is not a straightforward task. This paper aims to provide a simple manner to implement reliable Java code using CAA concepts. To do this, a Java framework (in Java-SE) that implements a formal coordination model based on CAA is presented. In addition, we have defined, in Java Pathfinder (JPF) model checker, CAA properties regarding systems exceptional behavior. Then, using both the framework and JPF, software developers can quickly implement the systems coordination of exceptional behavior, via instantiation of the framework, and formally check the predefined exceptional behavior properties on code (using JPF).

A.1.2 Year: 2013

Title: *"Generation of Java Programs Properties from Test Purposes"*
Authors: Simone Hanazumi and Ana C. V. de Melo
Published in: Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2013) - Doctoral Symposium
Date: July 2013

Abstract:
Software formal verification is a technique used to ensure that computational systems have high-quality and work properly. From the system specification described in a formal language, we define properties that must be satisfied during system execution to guarantee the software quality. Then, these properties should be implemented in a verifier, tool responsible for running the verification and for notifying which properties were satisfied or not. When the verification process finishes, the verifier will indicate to software developers the possible location of each code fault in the system. The disadvantages of using formal verification are the high cost to apply this technique in real systems, and the necessity of having people with experience in formal languages and formal methods. In addition, the implementation of properties related to a particular system in a verifier is a complex task.
To help software developers in the application of formal verification in Java programs, this work proposes the generation of properties code, for direct use in a verifier, from test purposes derived from system formal requirements.

A.1.3 Year: 2014

Title: *"Exercising Java Exceptions Using Java Pathfinder and Program Instrumentation"*
Authors: Alexandre L. Martins, Simone Hanazumi and Ana C. V. de Melo
Published in: Proceedings of the 14th International Conference on Computational Science and Its Applications (ICCSA 2014)
Date: June 2014

Abstract:
This paper presents an instrumentation technique to exercise the exceptional behavior of Java programs using the Java Pathfinder model checker. First, programs statements that are potentially able to throw exceptions are identified, and then a set of Java instructions are inserted into programs source code to throw exceptions, producing an instrumented program. Java Pathfinder is applied over this instrumented code to perform multiple executions of the system under verification and cover the program exceptional behavior. Additionally, our technique proposes a prototype of a new Java Pathfinder class to verify and test Java exceptions.

⋆ ⋆ ⋆

Title: *"Testing Java Exceptions: An Instrumentation Technique"*
Authors: Alexandre L. Martins, Simone Hanazumi and Ana C. V. de Melo
Published in: Proceedings of the 38th Annual International Computers, Software & Applications Conference (COMPSAC 2014)
Date: July 2014

Abstract:
Quality can be defined as the level of adequacy between the final product and its specification. Software industries use many testing methodologies to assure the high-quality of their software. Code coverage is one of these methodologies usually applied to test the expected (normal) behavior of software. Exception handling structures mainly appear in software code to deal with

problems caused by unexpected behavior. Because of this, they are not completely covered with the techniques for testing programs normal behavior. To overcome this limit, we present the Verify Ex, a Java class prototype, together with a Java source code instrumentation technique to exercise exception handling structures. The result is an increase of code coverage rate due to the inclusion of programs exceptional behavior in the testing execution and code coverage analysis.

⋆ ⋆ ⋆

Title: *"From Test Purposes to Formal JPF Properties"*
Authors: Simone Hanazumi, Ana C. V. de Melo and Corina S. Păsăreanu
Published in: Proceedings of the Java PathFinder Workshop (JPF Workshop 2014)
Date: November 2014

Abstract:
Software quality is traditionally achieved by applying testing techniques and/or formal verification to the system under development. Formal testing techniques have been defined to assure the software quality based on a set of test purposes the users want to observe on the system under testing (SUT). This paper proposes the use of Java PathFinder (JPF) to observe test purposes, defined as properties, over the whole program behavior. To make this approach available to test community, a user-friendly interface to define test purposes as properties is provided, together with a JPF extension to observe the test purposes. To define a property, the user only needs to select a property pattern, its scope, and the events of interest. Then, a finite-state automaton is generated and can be submitted to JPF to verify the defined property. As a result, JPF provides a report including a property verdict (pass/fail), and the program traces and percentage of the traversed automaton during the verification process. This coverage information can be used to analyze the performed verification and guide a complementary test suite generation.

A.1.4 Year: 2015

Title: *"A Classification of Test Purposes based on Testable Properties"*
Authors: Simone Hanazumi and Ana C. V. de Melo
Published in: Proceedings of the 15th International Conference on Computational Science and Its Applications (ICCSA 2015)
Date: June 2015

Abstract:
Test purposes are today key-elements for making the formal testing approach applicable to software. They are abstractions for a set of test cases to be observed in programs and some tools for formal testing concentrate their effort to apply test purposes to programs. Defining test purposes for actual systems, however, are not very straightforward. We must first realize what is the property behind the set of test cases we want to address. To help in defining the test purposes for a system under test, this paper presents a classification for test purposes based on the testable properties patterns. First, we analyze the meaning of properties patterns in the light of test purposes and then check the testability of these patterns. As a result, we provide a classification for testable patterns applicable to test purposes and users can choose which pattern better fits his/her set of test cases to be observed. Moreover, since test purposes are defined as standard properties, a model checker can be used to find executions for which test purposes can be satisfied.

⋆ ⋆ ⋆

Title: *"On the Testability of Properties Patterns"*
Authors: Simone Hanazumi and Ana C. V. de Melo

Published in: Proceedings of the 13th International Conference on Software Engineering and Formal Methods (SEFM 2015)
Date: September 2015

Abstract:
The specification pattern system (SPS) provides a simple methodology to specify program properties that can be used during software testing and verification. The testability concept establishes a connection between temporal properties and program traces to show which properties classes can actually be verified to reach a success/fail verdict. In this paper, we combine the SPS with the testability concept, showing that properties specified with certain patterns in global and local scopes are testable for programs with finite executions. This result implies that any property that is specified using this set of SPS patterns and scopes will receive a success/fail verdict when it is verified against a finite execution program by a model checker. In addition, we can analyze the program and property traces that are obtained in the verification process to extract data that can guide us in the test cases generation.

A.2 Papers Under Review

Title: *"Towards MC/DC coverage on properties specification patterns"*
Authors: Ana C. V. de Melo, Corina S. Păsăreanu and Simone Hanazumi
Submitted to: an International Conference in the field of Software Engineering

Abstract:
Model based testing is used to validate the actual system against its requirements described as formal specification, while formal verification proves that a requirement is not violated in the overall system. Verifying properties, in certain cases, becomes very expensive (or unpractical), mainly when the application of test techniques is enough for the users purposes. The *Modified Condition/Definition Coverage* (MC/DC), adopted in the avionics software industry, is recognized as a good technique to find the possible mistakes on programs logics because it covers how each event/variable can affect the program results. It has also been adapted to provide the coverage of specifications in the requirements-based approach.
This paper proposes a technique to decompose properties (specifications), defined as regular expressions, into subexpressions representing test cases to cover the MD/DC for specifications (Unique First Word Recognition). Then, instead of proving an entire property, we can use a model checker to observe and select program executions that cover all the test cases given as the subexpressions. To support this approach, we give a syntactic characterization of the properties decomposition, inductively defined over the syntax of regular expressions, and show how to use the technique to decompose Specification Patterns (SPS) and monitor their satisfiability using the Java PathFinder (JPF).

⋆ ⋆ ⋆

Title: *"OConGraX: : A tool for data-flow test requirements generation"*
Authors: Simone Hanazumi, Paulo R. A. F. Nunes and Ana C. V. de Melo
Submitted to: an International Journal in the field of Software Engineering

Abstract:
The more complex to develop and manage systems the more software design faults increase, making fault-tolerant systems highly required. In the object-oriented programming paradigm, exception-handling mechanisms provide a clear separation of codes for error recovery and normal behavior and have been used in fault-tolerant software. To ensure the quality of fault-tolerant systems, both the normal and exceptional behaviors must be tested and/or verified. However,

software testing is still a difficult and costly software development task. Although the great efforts employed to develop techniques for testing programs' normal behaviors, there is still a lack of techniques and tools to effectively test the exceptional behavior. In this context, one of the promising approaches is the data-flow testing strategy applied to objects and exception mechanisms.
This article presents a tool to help testing and analyzing fault-tolerant systems by providing an automatic generation of data-flow test requirements for objects and exception-handling mechanisms of Java programs.

⋆ ⋆ ⋆

Title: ***"A Formal Approach to Implement Java Exceptions in Cooperative Systems"***
Authors: Simone Hanazumi and Ana C. V. de Melo
Submitted to: an International Journal in the field of Software Engineering

Abstract:
The increasing number of systems that work on the top of cooperating elements have required new techniques to control cooperation on both normal and abnormal behaviors of systems. The controllability of the normal behaviors has received more attention because they are concerned with the users expectations, while for the abnormal behaviors it is left to designers and programmers. However, for cooperative systems, the abnormal behaviors, mostly represented by exceptions at programming level, become an important issue in software development because they can affect the overall system behavior. If an exception is raised and not handled accordingly, the system may collapse. To avoid such situation, certain concepts and models have been proposed to coordinate propagation and recovering of exceptional behaviors, including the coordinated atomic action (CAA). Regardless of the effort in creating these conceptual models, an actual implementation of them in real systems is not very straightforward.
This article provides a reliable framework for the implementation of Java exceptions propagation and recovery using CAA concepts. To do this, a Java framework (based on a formal specification) is presented, together with a set of properties to be preserved and proved with the Java Pathfinder (JPF) model checker. In practice, to develop new systems based on the given coordination concepts, designers/programmers can instantiate the framework to implement the exceptional behavior and then verify the correctness of the resulting code using JPF. Therefore, by using the framework, designers/programmers can reuse the provided CAA implementation and instantiate fault-tolerant Java systems.

B — Linear Temporal Logic (LTL)

Linear Temporal Logic or LTL [Pnu77, MP90, MP91, MP92, BK08] extends propositional logic by modalities that allow to refer to the behavior of reactive systems [BK08]. It considers, besides the inputs and the outputs of a computation, the executions of systems to analyze their correctness. In the next sections we briefly describe the LTL basic terms [BK08, MP90, MP91, MP92], syntax and semantics [BK08, MP91, MP92].

B.1 Basic Terms

Atomic Proposition. An atomic proposition is used to express simple known facts (e.g. assertions about the values of variables) about the states/events of the system under consideration. For example, give an integer variable x, "$x \leq 10$" is an atomic proposition.

Alphabet (2^{AP}). An alphabet in LTL is the power set of *AP* (set of atomic propositions), i.e., it is the set of all subsets of AP. This set of formulas over AP can be inductively defined by the following four rules [BK08]:

1. *true* is a formula;
2. Any atomic proposition $P \in AP$ is a formula;
3. If φ_1, φ_2 and φ are formulas, then so are $(\neg\varphi)$ and $(\varphi_1 \wedge \varphi_2)$;
4. Nothing else is a formula.

Language of a LTL formula φ. The language of φ, or $\mathsf{Words}(\varphi)$, is the set of all infinite words (infinite concatenations of symbols from the alphabet 2^{AP}) over the alphabet 2^{AP} that satisfies the LTL formula φ, i.e., to every LTL formula a single linear temporal property is associated.

Past Formula. LTL formula that contains no future operators.

Future Formula. LTL formula that contains no past operators.

State Formula. LTL formula in which the temporal operators are not used. A state formula is both a past and a future formula.

Temporal Formula. LTL formula that makes use of temporal operators, boolean operators and quantification.

Model. A model σ for a temporal formula φ is an infinite sequence of states/events $\sigma = s_0, s_1, ...$, where each s_j provides an interpretation for the variables mentioned in φ. This interpretation is used in the variables evaluation, in which a truth value (0 for "false", or 1 for "true") will be assigned to each variable in φ. Being the evaluation a function μ, we can define it formally as $\mu : AP \rightarrow \{0, 1\}$. In addition, we say that a model σ satisfies φ at a position $j \geq 0$ if the variables evaluation using the interpretation given by s_j makes the temporal formula *true*. Formally, the satisfaction relation $\models$ can be defined as a set of pairs (μ, φ) where μ is an evaluation and φ is a formula. It is written as $\mu \models \varphi$.

B.2 Syntax

LTL formulas over the set AP with $P \in AP$ are formed using the grammar that is presented next [BK08, MP92]:

Grammar

$$\begin{array}{rll} \varphi ::= & \mathsf{True} \mid \mathsf{False} \mid & \textit{(Boolean Variables)} \\ & P \mid & \textit{(Atomic Proposition)} \\ & \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \oplus \varphi_2 \mid \varphi_1 \rightarrow \varphi_2 \mid \varphi_1 \leftrightarrow \varphi_2 \mid \neg\varphi \mid & \textit{(Boolean Operators)} \\ & \bigcirc\varphi \mid \Box\varphi \mid \Diamond\varphi \mid \varphi_1 \,\mathcal{U}\, \varphi_2 \mid \varphi_1 \,\mathcal{W}\, \varphi_2 & \textit{(Temporal Operators)} \end{array}$$

Although we only presented the future temporal operators in the grammar, we could extend it to add the past temporal operators. In the next section we describe the semantics of each boolean and (past and future) temporal operators.

B.3 Semantics

To describe the LTL operators, we present both informal and formal definitions [BK08, MP92, End72]. The formal definition is the interpretation of that operator in a given model. It represents the notion of a formula φ holding at a position j, $j \geq 0$, in a sequence σ of states/events: $(\sigma, j) \models \varphi$.

B.3.1 Quantification

There are two quantifiers in LTL: *existential* ($\exists$) and *universal* ($\forall$). To give their semantics in LTL, consider two models over the alphabet 2^{AP}, $\sigma : s_0, s_1, \ldots$ and $\sigma' : s'_0, s'_1, \ldots$, and a variable u such that $u \in 2^{AP}$. The model σ' is a u-variant of σ if for each $j \geq 0$, s'_j is a u-variant of s_j, i.e., s'_j differs from s_j by at most the interpretation given to u. Using this definition of u-variant, we present the semantics of each quantifier below.

Definition B.3.1 — Existential ($\exists$). Let φ be a LTL expression. The formula "$\exists u : \varphi$" holds at position j if φ holds for some σ' that is a $u - variant$ of σ.

$$(\sigma, j) \models \exists u : \varphi \qquad \text{iff} \qquad (\sigma', j) \models \varphi \text{ for some } \sigma'\text{, a } u\text{-variant of } \sigma$$

Definition B.3.2 — Universal ($\forall$). Let φ be a LTL expression. The formula "$\forall u : \varphi$" holds at position j if φ holds for every σ' that is a $u - variant$ of σ.

$$(\sigma, j) \models \forall u : \varphi \qquad \text{iff} \qquad (\sigma', j) \models \varphi \text{ for every } \sigma'\text{, a } u\text{-variant of } \sigma$$

B.3.2 Boolean Operators

The unary operator for *negation* ($\neg$), and the binary operators for *conjunction* ($\wedge$), *disjunction* ($\vee$), *implication* ($\rightarrow$) and *equivalence* ($\leftrightarrow$) are the boolean operators that can be applied to LTL formulas. Their usual meanings in propositional logic are preserved in LTL.

Definition B.3.3 — Negation ($\neg$). Given a negation $\neg\varphi$, it states that $\neg\varphi$ holds at position j iff φ does not hold at position j. Formally, it is defined as:

$$(\sigma, j) \models \neg\varphi \quad \text{iff} \quad (\sigma, j) \not\models \varphi$$

Definition B.3.4 — Conjunction ($\wedge$). Given a conjunction $\varphi_1 \vee \varphi_2$, it states that $\varphi_1 \vee \varphi_2$ holds at position j iff φ_1 and φ_2 hold. Formally, it is defined as:

$$(\sigma, j) \models \varphi_1 \wedge \varphi_2 \quad \text{iff} \quad (\sigma, j) \models \varphi_1 \quad \text{and} \quad (\sigma, j) \models \varphi_2$$

Definition B.3.5 — Disjunction ($\vee$). Given a disjunction $\varphi_1 \vee \varphi_2$, it states that $\varphi_1 \vee \varphi_2$ holds at position j iff either φ_1 or φ_2 holds. Formally, it is defined as:

$$(\sigma, j) \models \varphi_1 \vee \varphi_2 \quad \text{iff} \quad (\sigma, j) \models \varphi_1 \quad \text{or} \quad (\sigma, j) \models \varphi_2$$

Definition B.3.6 — Exclusive Disjunction ($\oplus$). Given a exclusive disjunction $\varphi_1 \oplus \varphi_2$, it states that $\varphi_1 \oplus \varphi_2$ holds at position j iff φ_1 or φ_2 holds, but not both. Formally, it is defined as:

$$(\sigma, j) \models \varphi_1 \oplus \varphi_2 \quad \text{iff} \quad (\sigma, j) \models \varphi_1 \quad \text{xor} \quad (\sigma, j) \models \varphi_2$$

Definition B.3.7 — Implication ($\rightarrow$). Given an implication $\varphi_1 \rightarrow \varphi_2$, it states that $\varphi_1 \rightarrow \varphi_2$ holds at position j iff either $\neg\varphi_1$ or φ_2 holds. Formally, it is defined as:

$$(\sigma, j) \models \varphi_1 \rightarrow \varphi_2 \quad \text{iff} \quad (\sigma, j) \models \neg\varphi_1 \quad \text{or} \quad (\sigma, j) \models \varphi_2$$

Definition B.3.8 — Equivalence ($\leftrightarrow$). Given an equivalence $\varphi_1 \leftrightarrow \varphi_2$, it states that $\varphi_1 \leftrightarrow \varphi_2$ holds at position j iff $\varphi_1 \rightarrow \varphi_e$ and $\varphi_2 \rightarrow \varphi_1$ hold. Formally, it is defined as:

$$(\sigma, j) \models \varphi_1 \leftrightarrow \varphi_2 \quad \text{iff} \quad (\sigma, j) \models \varphi_1 \rightarrow \varphi_2 \quad \text{and} \quad (\sigma, j) \models \varphi_2 \rightarrow \varphi_1$$

Truth Tables

To illustrate when the formulas with these boolean operators are valid, we present the truth tables [End72] for each operator (Table B.1). The rightmost column presents whether the formula has a true value assigned to it or not, according to the values assigned to the formula's logical propositions presented in the left column(s).

(a) Negation

φ	$\neg\varphi$
True	False
False	True

(b) Conjunction

φ_1	φ_2	$\varphi_1 \wedge \varphi_2$
True	True	True
True	False	False
False	True	False
False	False	False

(c) Disjunction

φ_1	φ_2	$\varphi_1 \vee \varphi_2$
True	True	True
True	False	True
False	True	True
False	False	False

(d) Exclusive Disjunction

φ_1	φ_2	$\varphi_1 \vee \varphi_2$
True	True	False
True	False	True
False	True	True
False	False	False

(e) Implication

φ_1	φ_2	$\varphi_1 \rightarrow \varphi_2$
True	True	True
True	False	False
False	True	True
False	False	True

(f) Equivalence

φ_1	φ_2	$\varphi_1 \leftrightarrow \varphi_2$
True	True	True
True	False	False
False	True	False
False	False	True

Table B.1: *Truth Tables - Boolean Operators*

Equivalence Rules

In this section we present a few equivalence rules involving the boolean operators above [End72, BK08, O'R08]. Some of them are applied in the proof sketches of Appendix 4.

Definition B.3.9 — Identity.

1. $(\varphi \wedge \mathsf{True}) \equiv (\mathsf{True} \wedge \varphi) \equiv \mathsf{True}$
2. $(\varphi \vee \mathsf{False}) \equiv (\mathsf{False} \vee \varphi) \equiv \varphi$

Definition B.3.10 — Negation.

1. $(\neg(\neg\varphi)) \equiv \varphi$
2. $(\neg(\varphi_1 \rightarrow \varphi_2)) \equiv (\varphi_1 \wedge (\neg\varphi_2))$
3. $(\neg(\varphi_1 \leftrightarrow \varphi_2)) \equiv ((\varphi_1 \wedge (\neg\varphi_2)) \vee ((\neg\varphi_1) \wedge \varphi_2))$

Definition B.3.11 — Idempotency Laws.

1. $(\varphi \wedge \varphi) \equiv \varphi$
2. $(\varphi \vee \varphi) \equiv \varphi$

Definition B.3.12 — Absorption Laws.

1. $(\varphi_1 \wedge (\varphi_2 \vee \varphi_1)) \equiv \varphi_1$
2. $(\varphi_1 \vee (\varphi_2 \wedge \varphi_1)) \equiv \varphi_1$

Definition B.3.13 — Distributive Laws.

1. $(\varphi_1 \wedge (\varphi_2 \vee \varphi_3)) \equiv ((\varphi_1 \wedge \varphi_2) \vee (\varphi_1 \wedge \varphi_3))$
2. $(\varphi_1 \vee (\varphi_2 \wedge \varphi_3)) \equiv ((\varphi_1 \vee \varphi_2) \wedge (\varphi_1 \vee \varphi_3))$

Definition B.3.14 — De Morgan's Laws.

1. $(\neg(\varphi_1 \wedge \varphi_2)) \equiv ((\neg\varphi_1) \vee (\neg\varphi_2))$
2. $(\neg(\varphi_1 \vee \varphi_2)) \equiv ((\neg\varphi_1) \wedge (\neg\varphi_2))$

Definition B.3.15 — Others.

1. Excluded Middle: $(\varphi \vee \neg\varphi) \equiv \mathsf{True}$.
2. Contradiction: $\neg(\neg\varphi \wedge \varphi) \equiv \mathsf{True}$; or, $\neg\varphi \wedge \varphi \equiv \mathsf{False}$.

B.3.3 Future Temporal Operators

The LTL temporal operators that expresses the future are: the unary operators *next* ($\bigcirc$), *eventually* ($\Diamond$) and *always* ($\Box$); and, the binary operators *until* ($\mathcal{U}$) and *waiting-for* ($\mathcal{W}$). The semantics of these temporal modalities are given below.

Definition B.3.16 — Next ($\bigcirc$). Let φ be a LTL expression. The formula $\bigcirc\varphi$ holds at position j if φ holds in the next step of the execution of the system under consideration.

$$(\sigma, j) \models \bigcirc\varphi \quad \text{iff} \quad (\sigma, j+1) \models \varphi$$

Definition B.3.17 — Eventually ($\Diamond$). Let φ be a LTL expression. The formula $\Diamond\varphi$ holds at position j if now or eventually in the future φ holds.

$$(\sigma, j) \vDash \Diamond\varphi \qquad \text{iff} \qquad (\sigma, k) \vDash \varphi \text{ for some } k \geq j$$

Definition B.3.18 — Always ($\Box$). Let φ be a LTL expression. The formula $\Box\varphi$ holds at position j if now and forever in the future φ holds.

$$(\sigma, j) \vDash \Box\varphi \qquad \text{iff} \qquad (\sigma, k) \vDash \varphi \text{ for all } k \geq j$$

Definition B.3.19 — Until ($\mathcal{U}$). Let φ_1 and φ_2 be LTL formulas. The expression $\varphi_1 \,\mathcal{U}\, \varphi_2$ holds at position j, if there is some future moment in which φ_2 holds, and φ_1 holds at all moments until that future moment.

$$(\sigma, j) \vDash \varphi_1 \,\mathcal{U}\, \varphi_2 \qquad \text{iff} \quad \begin{array}{l}\text{there exists a } k \geq j\text{, such that } (\sigma, k) \vDash \varphi_2\text{, and for every } i,\\ j \leq i < k, (\sigma, i) \vDash \varphi_1\end{array}$$

Definition B.3.20 — Waiting-for ($\mathcal{W}$). Let φ_1 and φ_2 be LTL formulas. The expression $\varphi_1 \,\mathcal{W}\, \varphi_2$ holds at position j if φ_1 holds continuously either until the next occurrence of φ_2 or throughout the sequence. This operator is a weaker version of the "Until" operator.

$$(\sigma, j) \vDash \varphi_1 \,\mathcal{W}\, \varphi_2 \qquad \text{iff} \qquad (\sigma, j) \vDash \varphi_1 \mathcal{U} \varphi_2 \quad \text{or} \quad \Box\varphi_1$$

Distributive Laws

Manna and Pnueli [MP92] have stated the distributive laws for future operators. Considering the abbreviation $\varphi_1 \Leftrightarrow \varphi_2$ for $\Box(\varphi_1 \leftrightarrow \varphi_2)$, we have the following duals:

Definition B.3.21 — Distributive Laws (Future Operators).

1. $\Box(\varphi_1 \wedge \varphi_2) \Leftrightarrow \Box\varphi_1 \wedge \Box\varphi_2$
2. $\Diamond(\varphi_1 \vee \varphi_2) \Leftrightarrow \Diamond\varphi_1 \vee \Diamond\varphi_2$
3. $\varphi_1 \,\mathcal{U}\, (\varphi_2 \vee \varphi_3) \Leftrightarrow (\varphi_1 \,\mathcal{U}\, \varphi_2) \vee (\varphi_1 \,\mathcal{U}\, \varphi_3)$
4. $(\varphi_1 \wedge \varphi_2) \,\mathcal{U}\, \varphi_3 \Leftrightarrow (\varphi_1 \,\mathcal{U}\, \varphi_3) \wedge (\varphi_2 \,\mathcal{W}\, \varphi_3)$
5. $\varphi_1 \,\mathcal{W}\, (\varphi_2 \vee \varphi_3) \Leftrightarrow (\varphi_1 \,\mathcal{W}\, \varphi_2) \vee (\varphi_1 \,\mathcal{W}\, \varphi_3)$
6. $(\varphi_1 \wedge \varphi_2) \,\mathcal{W}\, \varphi_3 \Leftrightarrow (\varphi_1 \,\mathcal{W}\, \varphi_3) \wedge (\varphi_2 \,\mathcal{W}\, \varphi_3)$

Duality

Manna and Pnueli [MP92] have defined the duals for future operators. Considering the abbreviation $\varphi_1 \Leftrightarrow \varphi_2$ for $\Box(\varphi_1 \leftrightarrow \varphi_2)$, we have the following duals:

Definition B.3.22 — Duality (Future Operators).

1. $\neg\Box\varphi_1 \Leftrightarrow \Diamond\neg\varphi_1$
2. $\neg\Diamond\varphi_1 \Leftrightarrow \Box\neg\varphi_1$
3. $\neg(\varphi_1 \,\mathcal{U}\, \varphi_2) \Leftrightarrow (\neg\varphi_2) \,\mathcal{W}\, (\neg\varphi_1 \wedge \neg\varphi_2)$
4. $\neg(\varphi_1 \,\mathcal{W}\, \varphi_2) \Leftrightarrow (\neg\varphi_2) \,\mathcal{U}\, (\neg\varphi_1 \wedge \neg\varphi_2)$
5. $\neg \bigcirc \varphi_1 \Leftrightarrow \bigcirc\neg\varphi_1$

B.3.4 Past Temporal Operators

The LTL temporal operators that expresses the past are: the unary operators *previous* (●), *once* (⧫) and *has-always-been* (■); and, the binary operators *since* ($\mathcal{S}$) and *back-to* ($\mathcal{B}$). The semantics of these temporal modalities are given below.

Definition B.3.23 — Previous (●). Let φ be a LTL expression. The formula $●\varphi$ holds at position j if φ holds in the previous step of the execution of the system under consideration.

$$(\sigma, j) \vDash ●\varphi \qquad \text{iff} \qquad (j > 0) \quad \text{and} \quad (\sigma, j-1) \vDash \varphi$$

Definition B.3.24 — Once (⧫). Let φ be a LTL expression. The formula $⧫\varphi$ holds at position j if now or once in the past φ holds.

$$(\sigma, j) \vDash ⧫\varphi \qquad \text{iff} \qquad (\sigma, k) \vDash \varphi \text{ for some } k,\ 0 \le k \le j$$

Definition B.3.25 — Has-always-been (■). Let φ be a LTL expression. The formula $■\varphi$ holds at position j if φ holds at position j and all preceding positions.

$$(\sigma, j) \vDash ■\varphi \qquad \text{iff} \qquad (\sigma, k) \vDash \varphi \text{ for all } k,\ 0 \le k \le j$$

Definition B.3.26 — Since ($\mathcal{S}$). Let φ_1 and φ_2 be LTL formulas. The expression $\varphi_1 \,\mathcal{S}\, \varphi_2$ holds at position j, if φ_2 has happened in the past, and φ_1 held continuously from the last occurrence of φ_2 to the present.

$$(\sigma, j) \vDash \varphi_1 \,\mathcal{S}\, \varphi_2 \qquad \text{iff} \quad \text{there exists a } k,\ 0 \le k \le j, \text{ such that } (\sigma, k) \vDash \varphi_2, \text{ and for every } i,\ k < i \le j, (\sigma, i) \vDash \varphi_1$$

Definition B.3.27 — Back-to ($\mathcal{B}$). Let φ_1 and φ_2 be LTL formulas. The expression $\varphi_1 \,\mathcal{B}\, \varphi_2$ holds at position j if φ_1 holds continuously at all positions preceding and including the present, either to the last occurrence of φ_2 or to position 0. This operator is a weaker version of the "Since" operator.

$$(\sigma, j) \vDash \varphi_1 \,\mathcal{B}\, \varphi_2 \qquad \text{iff} \qquad (\sigma, j) \vDash \varphi_1 \mathcal{S} \varphi_2 \quad \text{or} \quad ■\varphi_1$$

Remark. An important remark about the past operators is that they do not add expressiveness for the LTL. Gabbay [Gab87] has demonstrated that all LTL formulas can be represented using only the future temporal operators.

B.4 Finite Trace Semantics

As stated in Section B.1, LTL usually works with a model σ containing infinite sequence of states/events. However, when dealing with model checking [BK08] in real applications that run for a limited time, we want to work with finite sequence of states/events [RH05].

Roşu and Havelund [RH05] established a finite trace semantics for future time LTL (or LTL with future temporal operators). We present the definitions of the finite trace semantics next [BK08, RH05]. considering that either states or events may occur.

Definition B.4.1 — Finite Trace ($\hat{\pi}$). A finite trace is an infinite stationary sequence of states/events in which the last state/event is repeated infinitely.

Definition B.4.2 — Suffix Trace ($\hat{\pi}_i$). Given any trace $\hat{\pi}$, $\hat{\pi}_i$ denotes the suffix trace that starts at position i, with positions starting at 1.

Definition B.4.3 — Function: *head* (Trace $\mapsto$ State/Event). This function returns the head state/event of a trace, i.e., the first state/event of the trace.

Definition B.4.4 — Function: *tail* (Trace $\mapsto$ Trace). This function returns the tail of a trace, which is the trace without its head state/event.

Definition B.4.5 — Function: *length* (Trace $\mapsto$ Natural Number). This function returns the length of a trace.

Definition B.4.6 — Satisfaction Relation ($\models$). The satisfaction relation defines when a trace $\hat{\pi}$ satisfies a formula φ, written $\hat{\pi} \models \varphi$, and is defined inductively over the structure of the formulas as follows, where P is any atomic proposition and φ_1 and φ_2 are any formulas:

$$\begin{array}{lll}
\hat{\pi} \models \mathtt{True} & \text{iff} & \mathtt{True}, \\
\hat{\pi} \models \mathtt{False} & \text{iff} & \mathtt{False}, \\
\hat{\pi} \models P & \text{iff} & P \in \mathit{head}(\hat{\pi}) \\
\hat{\pi} \models \varphi_1 \wedge \varphi_2 & \text{iff} & \hat{\pi} \models \varphi_1 \text{ and } \hat{\pi} \models \varphi_2 \\
\hat{\pi} \models \varphi_1 \oplus \varphi_2 & \text{iff} & \hat{\pi} \models \varphi_1 \text{ xor } \hat{\pi} \models \varphi_2 \\
\hat{\pi} \models \bigcirc\varphi_1 & \text{iff} & \text{if } \mathit{tail}(\hat{\pi})\text{is defined then } \mathit{tail}(\hat{\pi}) \models \varphi_1 \text{ else } \hat{\pi} \models \varphi_1, \\
\hat{\pi} \models \Diamond\varphi_1 & \text{iff} & (\exists i \leq \mathit{length}(\hat{\pi}))\ \hat{\pi}_i \models \varphi_1, \\
\hat{\pi} \models \Box\varphi_1 & \text{iff} & (\forall i \leq \mathit{length}(\hat{\pi}))\ \hat{\pi}_i \models \varphi_1, \\
\hat{\pi} \models \varphi_1\ \mathcal{U}\ \varphi_2 & \text{iff} & (\exists i \leq \mathit{length}(\hat{\pi}))\ \hat{\pi}_i \models \varphi_2 \text{ and } (\forall j > i)\hat{\pi}_j \models \varphi_1
\end{array}$$

B.4.1 Examples

In this section we present a set of examples to illustrate the use of the finite traces semantics [RH05].

■ **Example B.4.1** Consider a finite trace $\hat{\pi} = a, b, c$. The functions *head*, *tail* and *length* return the following values:

- *head* ($\hat{\pi}$) = a
- *tail* ($\hat{\pi}$) = b, c
- *length* ($\hat{\pi}$) = 3

■ **Example B.4.2** Consider a finite trace $\hat{\pi} = a, b, c, d, e$. For $i = 3$ we have that ($\hat{\pi}_i = c, d, e$); and, for $j = 5$ we have that ($\hat{\pi}_j = e$).

■ **Example B.4.3** Consider the formula $\Diamond(\Box P \vee \Box\neg P)$. It is not satisfied by infinite traces, but it is satisfied by any finite trace because the last state/event of the trace either contains P or it does not. The negation of this formula is obtained by applying Definitions B.3.22 (1, 2) and B.3.14(2): $\Box(\Diamond P \wedge \Diamond\neg P)$. It is satisfied by infinite traces, but it is not satisfied by any finite trace.

C — Prospec Formulas - Testability Analysis

Table C.1: *Prospec - LTL formulas for Patterns and Scopes [SGRE11]*

Pattern	Scope	LTL Formula
Absence (P)	Global	$\neg(\Diamond P)$
	Before R	$\Diamond R \rightarrow \neg(\neg R \,\mathcal{U}\, P)$
	After L	$\neg((\neg L) \,\mathcal{U}\, (L \wedge \Diamond P))$
	Between L and R	$\Box((L \wedge \neg R \wedge \Diamond R) \rightarrow \neg(\neg R \,\mathcal{U}\, P))$
	After L Until R	$\Box(L \wedge \neg R \rightarrow \neg(\neg R \,\mathcal{U}\, P))$
Existence (P)	Global	$\Diamond P$
	Before R	$\neg((\neg P) \,\mathcal{U}\, R)$
	After L	$\neg((\neg L) \,\mathcal{U}\, (L \wedge \neg\Diamond P))$
	Between L and R	$\Box((L \wedge \neg R) \rightarrow (\neg((\neg P) \,\mathcal{U}\, R)))$
	After L Until R	$\Box((L \wedge \neg R) \rightarrow (\neg R \,\mathcal{U}\, (P \wedge \neg R)))$
Universality (P)	Global	$\Box P$
	Before R	$\Diamond R \rightarrow (P \,\mathcal{U}\, R)$
	After L	$\neg L \,\mathcal{W}\, (L \wedge \Box P))$
	Between L and R	$\Box((L \wedge \neg R \wedge \Diamond R) \rightarrow (P \,\mathcal{U}\, R))$
	After L Until R	$\Box(L \wedge \neg R \rightarrow (P \,\mathcal{W}\, R))$
Precedence (T, P) (T Precedes P)	Global	$\neg((\neg T) \,\mathcal{U}\, (P \wedge \neg T))$
	Before R	$\Diamond R \rightarrow (\neg P \,\mathcal{U}\, (T \vee R))$
	After L	$\neg((\neg L) \,\mathcal{U}\, (L \wedge ((\neg T) \,\mathcal{U}\, (P \wedge \neg T))))$
	Between L and R	$\Box((L \wedge \neg R \wedge \Diamond R) \rightarrow (\neg P \,\mathcal{U}\, (T \vee R)))$
	After L Until R	$\Box((L \wedge \neg R) \rightarrow$ $(\neg(((\neg T) \wedge \neg R) \,\mathcal{U}\, (P \wedge (\neg T) \wedge \neg R))))$
Response (T, P) (T Responds to P)	Global	$\Box(P \rightarrow \Diamond T)$
	Before R	$\neg((\neg R) \,\mathcal{U}\, (P \wedge (\neg R) \wedge ((\neg T) \,\mathcal{U}\, R)))$
	After L	$\neg((\neg L) \,\mathcal{U}\, (L \wedge (\neg\Box(P \rightarrow \Diamond T))))$
	Between L and R	$\Box((L \wedge \neg R) \rightarrow \neg((\neg R) \,\mathcal{U}\, (P \wedge (\neg R) \wedge ((\neg T) \,\mathcal{U}\, R))))$
	After L Until R	$\Box((L \wedge \neg R) \rightarrow \neg((\neg R) \,\mathcal{U}$ $(P \wedge (\neg R) \wedge ((\Box((\neg T) \wedge \neg R)) \vee ((\neg T) \,\mathcal{U}\, R)))))$

Continuing the study of testable properties in the SPS local scopes (Chapters 3 and 4), we analyzed the Prospec formulas (Table C.1)[SGRE11] to certify that, besides keeping the properties testability, they have the same semantics of the SPS original formulas [Pat]. [1].

Similarly to the study of Chapter 4, this study covers the SPS patterns that are able to represent the testable properties classes:safety class (patterns: *universality*, *absence*, *precedence*); guarantee class (pattern: *existence*); and recurrence class (pattern: *response*). For each pattern, we analyzed the LTL formulas in the scopes *before R*, *after L*, *between L and R*, and *after L until R*, using the definitions provided by LTL syntax and semantics (Sections B.2 and B.3). The same statements that were used in the previous study are applied here.

C.1 Safety Properties: Universality Pattern

The universality pattern states that a given state/event occurs throughout a scope. It is a safety property, since in the global scope it corresponds to the canonical formula $\Box P$, where P represents the state/event that must hold. Table C.2 presents the LTL formulas for this pattern. The ✓ symbol indicates that other works have already shown that the formula preserves the safety property (and therefore it is testable) in the given scope [NGH93, FFJ+10]; the ► symbol indicates the formulas that we analyze in the next sections to show that they also preserve the safety property in the given scope; and the ≡ symbol indicates that the formulas of the same row are equivalents.

Table C.2: *LTL formulas for universality pattern [DAC99, SGRE11]*

Pattern	Scope		SPS		Prospec
Universality (P)	Global	✓	$\Box(P)$	≡	$\Box P$
	Before R	►	$\Diamond R \rightarrow (P \,\mathcal{U}\, R)$	≡	$\Diamond R \rightarrow (P \,\mathcal{U}\, R)$
	After L	►	$\Box(L \rightarrow \Box(P))$	►	$\neg L \,\mathcal{W}\, (L \wedge \Box P))$
	Between L and R	►	$\Box((L \wedge \neg R \wedge \Diamond R) \rightarrow (P \,\mathcal{U}\, R))$	≡	$\Box((L \wedge \neg R \wedge \Diamond R) \rightarrow (P \,\mathcal{U}\, R))$
	After L Until R	►	$\Box(L \wedge \neg R \rightarrow (P \mathcal{W} R))$	►	$\Box(L \wedge \neg R \rightarrow (P \,\mathcal{W}\, R))$

Before we proceed to the scopes analysis, we give one recurring definition in the proof sketches for this pattern.

Definition C.1.1 — $P \,\mathcal{U}\, R$. Consider a sequence σ of states $s_0, s_1, \ldots s_n$, where s_0 is the initial state and s_n is the final state of the program execution, and a position j such that $0 \leq j \leq n$. We apply the Definition B.3.19 and get the statement below:

$$(\sigma, j) \vDash P \,\mathcal{U}\, R \quad \text{iff} \quad \exists k,\ j \leq k \leq n,\ \text{s.t. } (\sigma, k) \vDash R \text{ and } \forall i,\ j \leq i < k,\ (\sigma, i) \vDash P$$

This expression states that *P* holds from position *j* until *R* occurs at *k*.

⇒ *The Definition C.1.1 is used in the Proof-Sketch C.1.2.*

C.1.1 Before R

Proposition C.1.1 — Prospec - Safety Property (Universality Pattern) in the Before R Scope. The *safety* property represented by the *universality* pattern is preserved in the *before R* scope.

The universality pattern in the *before R* scope is represented by the same formula in the SPS and the Prospec. Therefore, the proof sketch is the same of Proposition 4.1.1 (Chapter 4).

[1]The LTL formulas used by the Prospec are slightly different from the SPS original formulas (Table 2.3) because they were written to generate automata with a reduced number of states [SGRE11].

C.1.2 After L

Proposition C.1.2 — Prospec - Safety Property (Universality Pattern) in the After L Scope. The *safety* property represented by the *universality* pattern is preserved in the *after L* scope.

The universality pattern in the *after L* scope is represented by different formulas in the SPS and the Prospec. Therefore, we present below the proof sketch for the Prospec formula to show the validity of Proposition C.1.2.

Prospec Formula

The analyzed Prospec formula is the formula presented in Table C.1.

$$\neg((\neg L)\ \mathcal{U}\ (L \wedge \Diamond\neg P))$$

Proof Sketch C.1.1. Consider a sequence σ of states $s_0, s_1, \ldots s_n$, where s_0 is the initial state and s_n is the final state of the program execution. The following equivalence must hold:

$$\begin{array}{rcll} \neg((\neg L)\ \mathcal{U}\ (L \wedge \Diamond\neg P)) & \equiv & (\neg(L \wedge \neg\Box P))\ \mathcal{W}\ (L \wedge (\neg(L \wedge \neg\Box P))) & \text{Definition B.3.22 (1;3)} \\ & \equiv & (\neg L \vee \Box P)\ \mathcal{W}\ (L \wedge (\neg L \vee \Box P)) & \text{Definition B.3.14 (1)} \\ & \equiv & (\neg L \vee \Box P)\ \mathcal{W}\ ((L \wedge (\neg L)) \vee (L \wedge (\Box P))) & \text{Definition B.3.13 (1)} \\ & \equiv & (\neg L \vee \Box P)\ \mathcal{W}\ \mathtt{False} \vee (L \wedge (\Box P))) & \text{Definition B.3.15 (2)} \\ & \equiv & (\neg L \vee \Box P)\ \mathcal{W}\ (L \wedge (\Box P)) & \text{Definition B.3.9 (2)} \end{array}$$

Then, analyzing the formula $(\neg L \vee \Box P)\ \mathcal{W}\ (L \wedge (\Box P))$ we have that the following statement must hold, according to Definition B.3.20:

$$(\sigma, 0) \vDash (\neg L \vee \Box P)\ \mathcal{W}\ (L \wedge (\Box P)) \quad \text{iff} \quad (\sigma, 0) \vDash \Box(\neg L \vee \Box P) \quad \text{or} \quad (\neg L \vee \Box P)\ \mathcal{U}\ (L \wedge (\Box P))$$

We must analyze the two cases:

- **Case 1.** The formula $\Box(\neg L \vee \Box P)$ states that either $\neg L$ holds at every position of σ or P holds at every position of σ (Definitions B.3.5 and B.3.18). In this case, the scope is only defined when at some position k, $0 \leq k \leq n$ we have that $(\sigma, k) \vDash L \wedge \Box P$ (i.e. L occurs at k and P holds for all m s.t. $k \leq m \leq n$).
- **Case 2.** The formula $(\neg L \vee \Box P)\ \mathcal{U}\ (L \wedge (\Box P))$ states that either $\neg L$ or $\Box P$ holds until $L \wedge (\Box P)$ holds (Definition B.3.19). Since the states/events that take place before L can be ignored in this scope definition, we only analyze the formula $(L \wedge (\Box P))$ that must hold at some position k, s.t. $0 \leq k \leq n$. This formula states that L occurs at k and P holds for all m s.t. $k \leq m \leq n$.

Considering the two cases, we have that the safety property is preserved in *case 2*, and, in *case 1*, we have the scope definition in at least one situation. Therefore, we conclude that the safety property is preserved in the Prospec formula of *after L* scope. ■

C.1.3 Between L and R

Proposition C.1.3 — Prospec - Safety Property (Universality Pattern) in the Between L and R Scope. The *safety* property represented by the *universality* pattern is preserved in the *between L and R* scope.

The universality pattern in the *between L and R* scope is represented by the same formula in the SPS and the Prospec. Therefore, the proof sketch is the same of Proposition 4.1.3 (Chapter 4).

C.1.4 After L Until R

Proposition C.1.4 — Prospec - Safety Property (Universality Pattern) in the After L Until R Scope. The *safety* property represented by the *universality* pattern is preserved in the *after L until R* scope.

The universality pattern in the *after L until R* scope is represented by different formulas in the SPS and the Prospec. Therefore, we present below the proof sketch for the Prospec formula to show the validity of Proposition C.1.4.

Prospec Formula

The analyzed Prospec formula is the formula presented in Table C.1.

$$\Box Q, \text{ where } Q = (L \wedge \neg(R) \rightarrow (\neg((P \wedge \neg R)\ \mathcal{U}\ ((\neg P) \wedge \neg R))))$$

Proof Sketch C.1.2. Consider a sequence σ of states $s_0, s_1, \dots s_n$, where s_0 is the initial state and s_n is the final state of the program execution, and a position j such that $0 \leq j \leq n$. We first analyze the Q statement, using Definition B.3.7:

$$(\sigma, j) \vDash L \wedge \neg(R) \rightarrow (\neg((P \wedge \neg R)\ \mathcal{U}\ ((\neg P) \wedge \neg R)))$$

Using this statement, we can analyze each part of the implication as follows:

- **Case 1.** $(\sigma, j) \vDash L \wedge \neg R$ iff $(\sigma, j) \vDash L$ and $(\sigma, j) \vDash \neg R$. This formula deals with the states/events that limit the scope, and states that L occurs at position j and R does not.
- **Case 2.** The second part of the implication can be rewritten using some LTL definitions:

$\neg((P \wedge \neg R)\ \mathcal{U}\ ((\neg P) \wedge \neg R))$	
$\equiv (\neg(\neg P \wedge \neg R))\ \mathcal{W}((\neg(P \wedge \neg R)) \wedge (\neg(\neg P \wedge \neg R)))$	Definition B.3.22 (3)
$\equiv (P \vee R)\ \mathcal{W}\ ((\neg P \vee R) \wedge (P \vee R))$	Definition B.3.14 (1)
$\equiv (P \vee R)\ \mathcal{W}\ (R \vee (\neg P \wedge P))$	Definition B.3.13 (2)
$\equiv (P \vee R)\ \mathcal{W}\ (R \vee \mathtt{False})$	Definition B.3.15 (2)
$\equiv (P \vee R)\ \mathcal{W}\ R$	Definition B.3.9 (2)

  Thus, we can analyze the statement $(\sigma, j) \vDash (P \vee R)\ \mathcal{W}\ R$ iff $(\sigma, j) \vDash (P \vee R)\ \mathcal{U}\ R$ or $\Box(P \vee R)$ (Definition B.3.20) as follows:

  1. $(\sigma, j) \vDash (P \vee R)\ \mathcal{U}\ R$. This formula states that $P \vee R$ holds until the occurrence or R (Definition B.3.19). Then, we can analyze two situations:
     (a) $P\ \mathcal{U}\ R$: it is the Definition C.1.1 with k defined in the interval $j < k \leq n$. We have to avoid the situation $k = j$ because we have from *case 1* that $\neg R$ holds at position j; and, if $k = j$ then R would occur at position j either, leading us to a contradiction.
     (b) If there is another occurrence of R at some position $m, j \leq m < k$ the scope definition would be incomplete since the scope would be limited by the occurrence of R at position m. Hence, the disjunction $(P \vee R)$ cannot have a *true* value assigned to R (i.e. only P can occur).
  2. $(\sigma, j) \vDash \Box(P \vee R)$ iff $(\sigma, k) \vDash (P \vee R)$ for all k, s.t. $j \leq k \leq n$. This formula says that either P or R must hold from position j to n. So, the following situations can occur, according to the Table B.1 (c):
     (a) P is *true* and R is *false*: it states that $(\sigma, j) \vDash \Box P$, i.e., P holds from position j to n.
     (b) P is *false* and R is *true*: in this case P never occurs.
     (c) P is *true* and R is *true*: it states that P and R can occur in a non-ordered manner. In this case the scope is defined when the first occurrence of R, which limits the scope, takes place right after an occurrence of P.

Combining the results from *case 1* (where *situation 1(a) is valid*) and *case 2* (where *situations 2(a) and 2(c) are valid*), we reach the conclusions described next:

- L holds at position j and R does not;
- If there are no occurrences of R then P holds for all $i, j \leq i \leq n$;
- If R occurs at position k, $j < k \leq n$, P holds for all $i, j \leq i < k$.

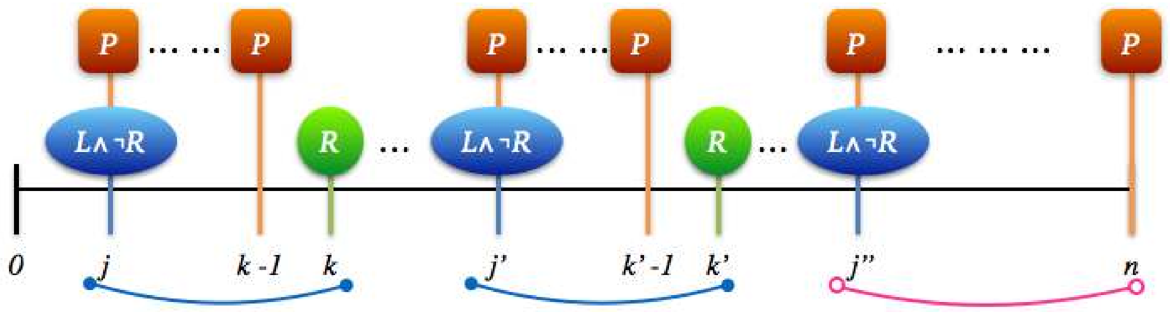


Figure C.1: *Universality pattern - After L until R scope (Prospec) - Scope Repetition*

Therefore we have that P holds for all positions in this scope delimited by L and R, or just L, and so the safety property is preserved in the Q statement (Figure C.1). Since the universality pattern is represented by $\square Q$ in this scope, we have that $(\sigma, j) \vDash \square Q$ iff $(\sigma, l) \vDash Q$,$\forall l,\ j \leq l \leq n$ (Definition B.3.18). Hence, $\square Q$ assures that the safety property represented by Q always occurs in the computation.

■

C.2 Safety Properties: Absence Pattern

The absence pattern states that a given state/event does not occur throughout a scope. It is a safety property, since in the global scope it corresponds to the formula $\square\neg P$, where P represents the state/event that cannot hold. Table C.3 presents the LTL formulas for this pattern. The ✓ symbol indicates that other works have already shown that the formula preserves the safety property (and therefore it is testable) in the given scope [NGH93, FFJ+10]; the ► symbol indicates the formulas that we analyze in the next sections to show that they also preserve the safety property in the given scope; and the ≡ symbol indicates that the formulas of the same row are equivalents.

Table C.3: *LTL formulas for absence pattern [DAC99, SGRE11]*

Pattern	Scope	SPS	Prospec
Absence (P)	Global	✓ $\square(\neg P)$	≡ $\neg(\diamond P)$
	Before R	► $\diamond R \rightarrow (\neg P\ \mathcal{U}\ R)$	► $\diamond R \rightarrow \neg(\neg R\ \mathcal{U}\ P)$
	After L	► $\square(L \rightarrow \square(\neg P))$	► $\neg((\neg L)\ \mathcal{U}\ (L \wedge \diamond P))$
	Between L and R	► $\square((L \wedge \neg R \wedge \diamond R) \rightarrow (\neg P\ \mathcal{U}\ R))$	► $\square((L \wedge \neg R \wedge \diamond R) \rightarrow \neg(\neg R\ \mathcal{U}\ P))$
	After L Until R	► $\square(L \wedge \neg R \rightarrow (\neg P \mathcal{W} R))$	► $\square(L \wedge \neg R \rightarrow \neg(\neg R\ \mathcal{U}\ P))$

Before we proceed to the scopes analysis, we give one recurring definition in the proof sketches for this pattern.

Definition C.2.1 — $\neg(P)\ \mathcal{W}\ (R \wedge \neg P)$. Consider a sequence σ of states $s_0, s_1, \ldots s_n$, where s_0 is the initial state and s_n is the final state of the program execution, and a position j such that $0 \leq j \leq n$. We apply the Definition B.3.19 and get the statement :

$$(\sigma, j) \vDash \neg(P)\ \mathcal{W}\ (R \wedge \neg P) \quad \text{iff} \quad (\sigma, j) \vDash \neg P\ \mathcal{U}\ (R \wedge \neg P) \text{ or } \square\neg P$$

Then at least one of the following statements should hold in the computation:

- $(\sigma, j) \vDash \neg P\ \mathcal{U}\ (R \wedge \neg P)$ iff $\exists k,\ j \leq k \leq n,$ s.t. $(\sigma, k) \vDash (R \wedge \neg P)$ and $\forall i,\ j \leq i < k,\ (\sigma, i) \vDash \neg P$ (Definition B.3.19). This expression states that $\neg P$ holds from position j until R occurs at k, and $\neg P$ also holds at k.

- $(\sigma, j) \models \Box\neg P \quad \text{iff} \quad (\sigma, k) \models \neg P \; \forall k \; j \leq k \leq n$ (Definition B.3.18). This expression states that $\neg P$ holds continuously for all k, s.t. $j \leq k \leq n$.

⇒ The Definition C.2.1 is used in Proof-Sketches C.2.1, C.2.3 and C.2.4.

Remark. The expression $(\sigma, j) \models \neg P \; \mathcal{U} \; (R \wedge \neg P)$ states that $\neg P$ holds from position j until R occurs at k, and $\neg P$ also holds at k. In the scope definitions limited by R we consider only the fact that $\neg P$ holds from position j until R occurs at k. The fact that $\neg P$ and R hold at k does not change the scopes semantics significantly since the states/events that take place at the same position of R are not included in the intervals of the scopes [SGRE11] (Section 2.4.1).

C.2.1 Before R

Proposition C.2.1 — Prospec - Safety Property (Absence Pattern) in the Before R Scope. The *safety* property represented by the *absence* pattern is preserved in the *before R* scope.

The absence pattern in the *before R* scope is represented by different formulas in the SPS and the Prospec. Therefore, we present below the proof sketch for the Prospec formula to show the validity of Proposition C.2.1.

Prospec Formula

The analyzed Prospec formula is the formula presented in Table C.1.

$$\Diamond R \rightarrow \neg(\neg(R) \; \mathcal{U} \; P) \equiv \Diamond R \rightarrow \neg(P) \; \mathcal{W} \; (R \wedge \neg P) \qquad \text{Definition B.3.22 (3)}$$

Proof Sketch C.2.1. Consider a sequence σ of states $s_0, s_1, \ldots s_n$, where s_0 is the initial state and s_n is the final state of the program execution. The following statement must hold:

$$(\sigma, 0) \models \Diamond R \rightarrow \neg(P) \; \mathcal{W} \; (R \wedge \neg P) \quad \text{therefore} \quad \text{if } (\sigma, 0) \models \Diamond R \text{ then } (\sigma, 0) \models \neg(P) \; \mathcal{W} \; (R \wedge \neg P)$$

Using this statement, we can analyze each part of the implication as follows:

- **Case 1.** $(\sigma, 0) \models \Diamond R \quad \text{iff} \quad (\sigma, k) \models R$ for some k, $0 \leq k \leq n$ (Definition B.3.17). This formula states that R occurs at position k, thus, the scope limit is defined;
- **Case 2.** It is the Definition C.2.1 for $j = 0$.

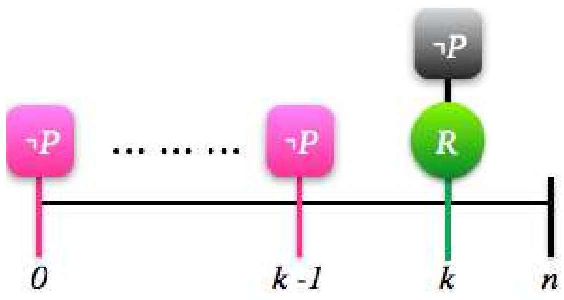


Figure C.2: *Absence pattern - Before R scope (Prospec)*

Analyzing both parts of the implication, we have that if R occurs at k, $0 \leq k \leq n$ then $\neg P$ occurs for all positions i, $0 \leq i < k$. In addition, $\neg P$ may also occur at k but it does not change the semantics of the scope since the states/events that takes place at the same position of R are not considered. Hence, the safety property is preserved in this scope.

■

C.2.2 After L

Proposition C.2.2 — Prospec - Safety Property (Absence Pattern) in the After L Scope. The *safety* property represented by the *absence* pattern is preserved in the *after L* scope.

The absence pattern in the *after L* scope is represented by different formulas in the SPS and the Prospec. Therefore, we present below the proof sketch for the Prospec formula to show the validity of Proposition C.2.2.

Prospec Formula

The analyzed Prospec formula is the formula presented in Table C.1.

$$\neg((\neg L)\ \mathcal{U}\ (L \wedge \Diamond P))$$

Proof Sketch C.2.2. Same as the Proof Sketch C.1.1, replacing $\neg P$ for P in the analyzed formula. ∎

C.2.3 Between L and R

Proposition C.2.3 — Prospec - Safety Property (Absence Pattern) in the Between L and R Scope. The *safety* property represented by the *absence* pattern is preserved in the *between L and R* scope.

The absence pattern in the *between L and R* scope is represented by different formulas in the SPS and the Prospec. Therefore, we present below the proof sketch for the Prospec formula to show the validity of Proposition C.2.3.

Prospec Formula

The analyzed Prospec formula is the formula presented in Table C.1.

$$\begin{aligned} \Box Q,\ \text{where } Q &= (L \wedge \neg R \wedge \Diamond R) \rightarrow \neg(\neg(R)\ \mathcal{U}\ P) \\ &\equiv (L \wedge \neg R \wedge \Diamond R) \rightarrow \neg P\ \mathcal{W}\ (R \wedge \neg P) \quad \text{(Definition B.3.22 (3))} \end{aligned}$$

Proof Sketch C.2.3. Consider a sequence σ of states $s_0, s_1, \ldots s_n$, where s_0 is the initial state and s_n is the final state of the program execution, and a position j such that $0 \leq j \leq n$. We first analyze the Q statement, using Definition B.3.7:

$$(\sigma, j) \models (L \wedge \neg R \wedge \Diamond R) \rightarrow (\neg P\ \mathcal{W}\ (R \wedge \neg P)) \quad \text{therefore} \quad \begin{array}{l} \text{if } (\sigma, j) \models L \wedge \neg R \wedge \Diamond R \\ \text{then } (\sigma, j) \models \neg P\ \mathcal{W}\ (R \wedge \neg P) \end{array}$$

Using this statement, we can analyze each part of the implication as follows:

- **Case 1.** $(\sigma, j) \models L \wedge \neg R \wedge \Diamond R$ iff $(\sigma, j) \models L$ and $(\sigma, j) \models \neg R$ and $(\sigma, j) \models \Diamond R$. This expression deals only with the occurrence of L and R, which are the states/events that limit the scope. It says that L occurs at position j and R does not. Moreover, we have that:
  - $(\sigma, j) \models \Diamond R$ iff $(\sigma, k) \models R$ for some $k > j$ (Definition B.3.17). The position k cannot be equal to position j, otherwise we would have at position j the occurrence of $\neg R \wedge R$, which is a contradiction. Therefore, we can assume that R occurs at position k, $j < k \leq n$.
- **Case 2.** Definition C.2.1.

Analyzing both parts of the implication, we have that if L occurs at j, $0 \leq j \leq n$, R does not occur at j; instead, R occurs at position k, $j < k \leq n$. Then, the model σ satisfies that $\neg P$ occurs for all positions i, $j \leq i < k$. Hence, the safety property is preserved in the interval defined by Q statement.

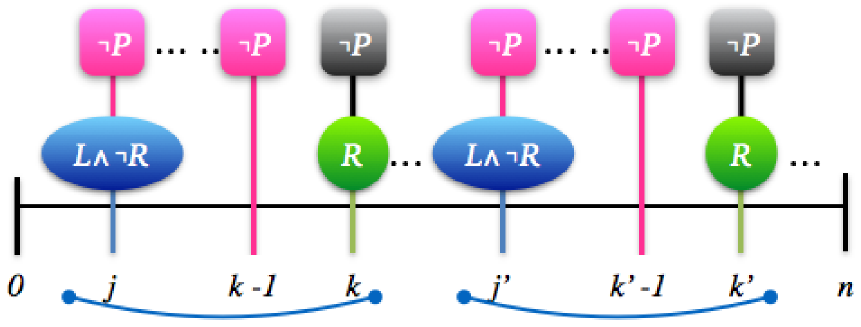


Figure C.3: *Absence pattern - Between L and R scope (Prospec) - Scope repetition*

Since the absence pattern is represented by $\Box Q$ in this scope, we have that $(\sigma, j) \vDash \Box Q \quad \text{iff} \quad (\sigma, l) \vDash Q, \forall l,\ j \leq l \leq n$ (Definition B.3.18). Hence, $\Box Q$ assures that the safety property represented by Q always occurs in the computation (Figure C.3).

■

C.2.4 After L Until R

Proposition C.2.4 — Prospec - Safety Property (Absence Pattern) in the After L Until R Scope. The *safety* property represented by the *absence* pattern is preserved in the *after L until R* scope.

The absence pattern in the *after L until R* scope is represented by different formulas in the SPS and the Prospec. Therefore, we present below the proof sketch for the Prospec formula to show the validity of Proposition C.2.4.

Prospec Formula

The analyzed Prospec formula is the formula presented in Table C.1.

$$\Box Q, \text{ where } Q = (L \wedge \neg R) \rightarrow \neg(\neg(R)\ \mathcal{U}\ P) \equiv (L \wedge \neg R) \rightarrow \neg P\ \mathcal{W}\ (R \wedge \neg P)$$

Proof Sketch C.2.4. Consider a sequence σ of states $s_0, s_1, \ldots s_n$, where s_0 is the initial state and s_n is the final state of the program execution, and a position j such that $0 \leq j \leq n$. We first analyze the Q statement, using Definition B.3.7:

$$(\sigma, j) \vDash L \wedge \neg R \rightarrow (P \mathcal{W} R) \quad \text{therefore} \quad \text{if } (\sigma, j) \vDash L \wedge \neg R \text{ then } (\sigma, j) \vDash (P\ \mathcal{W}\ R)$$

Using this statement, we can analyze each part of the implication as follows:

- **Case 1.** $(\sigma, j) \vDash L \wedge \neg R \quad \text{iff} \quad (\sigma, j) \vDash L$ and $(\sigma, j) \vDash \neg R$. Thus we can assume that L occurs at position j and R does not.
- **Case 2.** Definition C.2.1.

Analyzing both parts of the implication, we have that if L occurs at j, $0 \leq j \leq n$ and R does not occur at j, then, the model σ satisfies one of these two cases:

- $\neg P$ occurs $\forall i, j \leq i < k$ and R occurs at k, s.t. $j < k \leq n$.
- R does not occur and P holds $\forall i, j \leq i \leq n$. Note that in this case, R does not occur (anymore) until the end of computation.

Hence, since $\neg P$ holds continuously from the occurrence of L until a future occurrence of R or the end of computation, the safety property is preserved in the interval defined by Q statement.

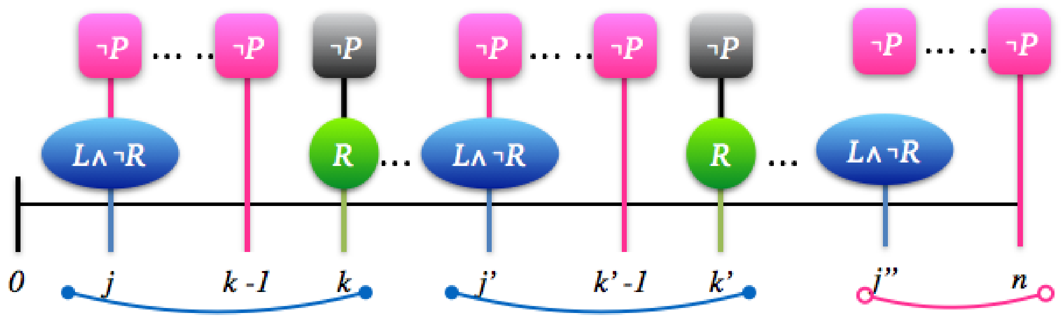


Figure C.4: *Absence pattern - After L Until R scope (Prospec) - Scope repetition*

Since the absence pattern is represented by $\square Q$ in this scope, we have that $(\sigma, j) \vDash \square Q$ iff $(\sigma, l) \vDash Q$, $\forall l,\ j \le l \le n$ (Definition B.3.18). Hence, $\square Q$ assures that the safety property represented by Q always occurs in the computation (Figure C.4).

■

C.3 Safety Properties: Precedence Pattern

The precedence pattern states that a state/event *P* must always be preceded by a state/event *T* within a scope. It is a safety property, since in the global scope it corresponds to the formula $\square(P \rightarrow \blacklozenge T) \equiv \neg P\ \mathcal{W}\ T$, where T represents the state/event that must precede the state/event P [MP92]. Table C.4 presents the LTL formulas for this pattern. The ✓ symbol indicates that other works have already shown that the formula preserves the safety property (and therefore it is testable) in the given scope [NGH93, FFJ$^+$10]; the ► symbol indicates the formulas that we analyze in the next sections to show that they also preserve the safety property in the given scope; and the $\equiv$ symbol indicates that the formulas of the same row are equivalents.

Table C.4: *LTL formulas for precedence pattern, [DAC99, SGRE11]*

Pattern	Scope	SPS	Prospec
Precedence (T, P)	Global	✓ $\neg P \mathcal{W} T$	$\equiv$ $\neg((\neg T)\ \mathcal{U}\ (P \wedge \neg T))$
	Before R	► $\diamond R \rightarrow (\neg P\ \mathcal{U}\ (T \vee R))$	$\equiv$ $\diamond R \rightarrow (\neg P\ \mathcal{U}\ (T \vee R))$
	After L	► $\square \neg L \vee \diamond(L \wedge (\neg P \mathcal{W} T))$	► $\neg((\neg L)\ \mathcal{U}\ (L \wedge ((\neg T)\ \mathcal{U}\ (P \wedge \neg T))))$
	Between L and R	► $\square((L \wedge \neg R \wedge \diamond R) \rightarrow (\neg P\ \mathcal{U}\ (T \vee R)))$	$\equiv$ $\square((L \wedge \neg R \wedge \diamond R) \rightarrow (\neg P\ \mathcal{U}\ (T \vee R)))$
	After L Until R	► $\square(L \wedge \neg R \rightarrow (\neg P \mathcal{W}(T \vee R)))$	► $\square((L \wedge \neg R) \rightarrow (\neg(((\neg T) \wedge \neg R)\ \mathcal{U}\ (P \wedge (\neg T) \wedge \neg R))))$

Remark. According to Manna and Pnueli [MP92], the following formulas represent the *precedence* pattern (safety property):

- $\square(P \rightarrow \blacklozenge T)$: this formula has the canonical form of a safety property (Section 3.2.2). It states that for each position j, if P holds at j, $0 \le j \le n$, there must be an earlier position k, $0 \le k \le j$ satisfying T (Definitions B.3.18 and B.3.24)[MP95].

- $\neg P\ \mathcal{W}\ T$: although it does not have the canonical form of a safety property, it keeps the semantics of a safety property. This formula states that either $\neg P$ holds forever or that it holds until an occurrence of T (Definition B.3.20). In other words, it says that the first occurrence of P must coincide or be preceded by a position in which T occurs [MP92].

Therefore, we have that $\square(P \rightarrow \blacklozenge T) \equiv \neg P\ \mathcal{W}\ T$.

C.3.1 Global Scope

Even though we have already shown that properties which are represented by the precedence pattern are testable in the global scope (Section 3.3.1), we could not neglect the fact that the new Prospec formula for this pattern (Table C.1) is different from the SPS original formula (Table 2.3). Then, we show in this section that the Prospec formula ($\neg((\neg T)\ \mathcal{U}\ (P \wedge \neg T))$) keeps the semantics of the SPS original formula ($\neg P \mathcal{W} T$):

Prospec Formula

$$\neg((\neg T)\ \mathcal{U}\ (P \wedge \neg T))$$

Proof Sketch C.3.1. Consider a sequence σ of states $s_0, s_1, \ldots s_n$, where s_0 is the initial state and s_n is the final state of the program execution. Analyzing the Prospec formula we get the following equivalent formula:

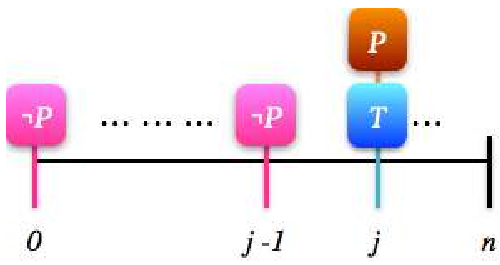
$$\begin{array}{rcll} \neg((\neg T)\ \mathcal{U}\ (P \wedge \neg T)) & \equiv & (\neg(P \wedge \neg T))\ \mathcal{W}\ (T \wedge (\neg(P \wedge \neg T))) & \text{Definition B.3.22 (3)} \\ & \equiv & (\neg P \vee T)\ \mathcal{W}\ (T \wedge (\neg P \vee T)) & \text{Definition B.3.14 (1)} \\ & \equiv & (\neg P \vee T)\ \mathcal{W}\ T & \text{Definition B.3.12 (1)} \end{array}$$

Then, we have that the following statement must hold:

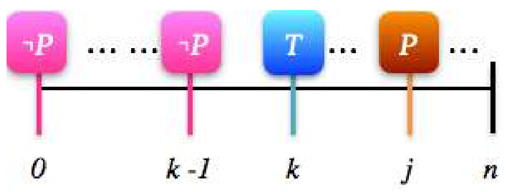
$$(\sigma, 0) \vDash (\neg P \vee T)\ \mathcal{W} T \quad \text{iff} \quad (\sigma, 0) \vDash (\neg P \vee T) \mathcal{U} T \quad \text{or} \quad \Box(\neg P \vee T)$$

We analyze each case separately:

- **Case 1.** $(\sigma, 0) \vDash \Box(\neg P \vee T)$ iff $(\sigma, k) \vDash (\neg P \vee T)\ \forall k, 0 \leq k \leq n$ (Definition B.3.18). This formula states that either $\neg P$ or T holds in the entire computation. The precedence pattern is preserved here because if T occurs at a position j then P holds at position j either. Moreover, if T does not occur then P does not occur in the whole computation.
- **Case 2.** $(\sigma, 0) \vDash (\neg P \vee T)\ \mathcal{U}\ T$. Here we have two possible situations:
  1. T holds at k, $0 \leq k \leq n$, $\neg P$ holds $\forall i$, $0 \leq i < k$ and P can hold at some position j, s.t. $k \leq j \leq n$.
  2. The (last) occurrence of T holds at k, $0 \leq k \leq n$, $\neg P$ and T occur in a non-ordered manner in the interval $[0, (k-1)]$, and P can hold at the same position of T or at some position j, s.t. $k \leq j \leq n$.



(a) T and P hold at j



(b) T precedes P

Figure C.5: *Precedence pattern - Global scope*

Combining the results from *cases 1 and 2* we have that the Prospec formula keeps the semantics of the SPS *precedence* pattern.

∎

C.3.2 Before R

Proposition C.3.1 — Prospec - Safety Property (Precedence Pattern) in the Before R Scope. The *safety* property represented by the *precedence* pattern is preserved in the *before R* scope.

The precedence pattern in the *before R* scope is represented by the same formula in the SPS and the Prospec. Therefore, the proof sketch is the same of Proposition 4.3.1 (Chapter 4).

C.3.3 After L

Proposition C.3.2 — Prospec - Safety Property (Precedence Pattern) in the After L Scope. The *safety* property represented by the *precedence* pattern is preserved in the *after L* scope.

The precedence pattern in the *after L* scope is represented by different formulas in the SPS and the Prospec. Therefore, we present below the proof sketch for the Prospec formula to show the validity of Proposition C.3.2.

Prospec Formula

The analyzed Prospec formula is the formula presented in Table C.1.

$$\neg((\neg L)\ \mathcal{U}\ (L \land ((\neg T)\ \mathcal{U}\ (P \land \neg T))))$$

Proof Sketch C.3.2. Consider a sequence σ of states $s_0, s_1, \ldots s_n$, where s_0 is the initial state and s_n is the final state of the program execution. To rewrite the Prospec formula, we use the following equivalence relation:

Definition C.3.1 — $\neg(L \land ((\neg T)\ \mathcal{U}\ (P \land \neg T))) \equiv \neg L \lor ((\neg P \lor T)\ \mathcal{W}\ T)$.

$$\begin{array}{ll}
(\neg(L \land ((\neg T)\ \mathcal{U}\ (P \land \neg T)))) & \\
\equiv (\neg L \lor \neg((\neg T)\ \mathcal{U}\ (P \land \neg T))) & \text{Definition B.3.14 (1)} \\
\equiv (\neg L \lor (\neg(P \land \neg T)\ \mathcal{W}\ (T \land \neg(P \land \neg T)))) & \text{Definition B.3.22 (3)} \\
\equiv (\neg L \lor ((\neg P \lor T)\ \mathcal{W}\ (T \land (\neg P \lor T)))) & \text{Definition B.3.14 (1)} \\
\equiv (\neg L \lor ((\neg P \lor T)\ \mathcal{W}\ T)) & \text{Definition B.3.12 (1)}
\end{array}$$

Now we use this equivalence (Formula C.3.1) and rewrite the Prospec formula as follows:

$$\begin{array}{ll}
\neg((\neg L)\ \mathcal{U}\ (L \land ((\neg T)\ \mathcal{U}\ (P \land \neg T)))) & \\
\equiv (\neg(L \land ((\neg T)\ \mathcal{U}\ (P \land \neg T)))\ \mathcal{W}\ (L \land (\neg(L \land ((\neg T)\ \mathcal{U}\ (P \land \neg T)))))) & \text{Definition B.3.22 (3)} \\
\equiv (\neg L \lor ((\neg P \lor T)\ \mathcal{W}\ T))\ \mathcal{W}\ (L \land (\neg L \lor ((\neg P \lor T)\ \mathcal{W}\ T))) & \text{Definition C.3.1} \\
\equiv (\neg L \lor ((\neg P \lor T)\ \mathcal{W}\ T))\ \mathcal{W}\ ((L \land \neg L) \lor (L \land ((\neg P \lor T)\ \mathcal{W}\ T))) & \text{Definition B.3.13 (1)} \\
\equiv (\neg L \lor ((\neg P \lor T)\ \mathcal{W}\ T))\ \mathcal{W}\ (\mathit{False} \lor (L \land ((\neg P \lor T)\ \mathcal{W}\ T))) & \text{Definition B.3.15 (2)} \\
\equiv (\neg L \lor ((\neg P \lor T)\ \mathcal{W}\ T))\ \mathcal{W}\ (L \land ((\neg P \lor T)\ \mathcal{W}\ T)) & \text{Definition B.3.9 (2)}
\end{array}$$

Thus, we can analyze the following statement:

$$\begin{array}{lll}
(\neg L \lor ((\neg P \lor T)\ \mathcal{W}\ T)) & \mathcal{W}\ (L \land ((\neg P \lor T)\ \mathcal{W}\ T)) & \\
& \text{iff} \quad \Box(\neg L \lor ((\neg P \lor T)\ \mathcal{W}\ T)) & \text{or} \\
& \quad (\neg \quad L \lor ((\neg P \lor T)\ \mathcal{W}\ T))\ \mathcal{U}\ (L \land ((\neg P \lor T)\ \mathcal{W}\ T)) &
\end{array}$$

Analyzing the two cases:

- **Case 1.** The formula $\Box(\neg L \lor ((\neg P \lor T)\ \mathcal{W}\ T))$ states that either $\neg L$ holds at every position of σ or $((\neg P \lor T)\ \mathcal{W}\ T)$ holds at every position of σ (Definitions B.3.5 and B.3.18). In this case, the scope is only defined when at some position k, $0 \leq k \leq n$ we have that $(\sigma, k) \vDash L \land ((\neg P \lor T)\ \mathcal{W}\ T)$ (i.e. L occurs at k and $((\neg P \lor T)\ \mathcal{W}\ T)$, which is the precedence pattern in the global scope (Table C.4), holds at k).

- **Case 2.** The formula $(\neg L \vee ((\neg P \vee T)\ \mathcal{W}\ T))\ \mathcal{U}\ (L \wedge ((\neg P \vee T)\ \mathcal{W}\ T))$ states that either $\neg L$ or $(\neg P \vee T)\ \mathcal{W}\ T$ holds until $L \wedge ((\neg P \vee T)\ \mathcal{W}\ T)$ holds. (Definition B.3.19). Since the states/events that take place before L can be ignored in this scope definition, we only analyze the formula $(L \wedge ((\neg P \vee T)\ \mathcal{W}\ T))$ that must hold at some position k, s.t. $0 \leq k \leq n$. Then, considering the execution at position k, we have that the following statements must hold:
  - The formula L holds at k, $0 \leq k \leq n$;
  - The formula $((\neg P \vee T)\ \mathcal{W}\ T)$, which is semantically equal to the SPS *precedence* pattern in the global scope (Proof Sketch C.3.1), holds at k, $0 \leq k \leq n$. This means that T occurs at some position i, $k \leq i \leq n$ and P occurs at some position j, $i \leq j \leq n$. i.e., T precedes P.

Considering the two cases, we have that the safety property is preserved in *case 2*, and in *case 1*, we have the scope definition in at least one situation. Therefore, we conclude that the safety property is preserved in the Prospec formula of *after L* scope. ■

C.3.4 Between L and R

Proposition C.3.3 — Prospec - Safety Property (Precedence Pattern) in the Between L and R Scope. The *safety* property represented by the *precedence* pattern is preserved in the *between L and R* scope.

The precedence pattern in the *between L and R* scope is represented by the same formula in the SPS and the Prospec. Therefore, the proof sketch is the same of Proposition 4.3.3 (Chapter 4).

C.3.5 After L Until R

Proposition C.3.4 — Prospec - Safety Property (Precedence Pattern) in the After L Until R Scope. The *safety* property represented by the *precedence* pattern is preserved in the *after L until R* scope.

The precedence pattern in the *after L until R* scope is represented by different formulas in the SPS and the Prospec. Therefore, we present below the proof sketch for the Prospec formula to show the validity of Proposition C.3.4.

Prospec Formula

The analyzed Prospec formula is the formula presented in Table C.1.

$$\Box Q, \text{ where } Q = (L \wedge \neg R) \rightarrow (\neg((\neg T \wedge \neg R)\ \mathcal{U}\ (P \wedge \neg T \wedge \neg R)))$$

Proof Sketch C.3.3. Consider a sequence σ of states $s_0, s_1, \ldots s_n$, where s_0 is the initial state and s_n is the final state of the program execution, and a position j such that $0 \leq j \leq n$. We first analyze the Q statement, using Definition B.3.7:

$$\begin{aligned}(\sigma, j) \models (L \wedge \neg R) \rightarrow\ & (\neg((\neg T \wedge \neg R)\ \mathcal{U}\ (P \wedge \neg T \wedge \neg R))) \quad \text{therefore}\\ & \text{if } (\sigma, j) \models L \wedge \neg R\\ & \text{then } (\sigma, j) \models (\neg((\neg T \wedge \neg R)\ \mathcal{U}\ (P \wedge \neg T \wedge \neg R)))\end{aligned}$$

Using this statement, we can analyze each part of the implication as follows:

- **Case 1.** $(\sigma, j) \models L \wedge \neg R$ iff $(\sigma, j) \models L$ and $(\sigma, j) \models \neg R$. This formula deals with the states/events that limit the scope, and states that L occurs at position j and R does not.

- **Case 2.** $(\sigma, j) \models (\neg((\neg T \wedge \neg R)\ \mathcal{U}\ (P \wedge \neg T \wedge \neg R)))$. This formula can be rewritten using some LTL definitions (Section B.3):

$$\begin{aligned}&\neg((\neg T \wedge \neg R)\ \mathcal{U}\ (P \wedge \neg T \wedge \neg R))\\ &\equiv \neg(P \wedge \neg T \wedge \neg R)\ \mathcal{W}\ (\neg(\neg T \wedge \neg R) \wedge \neg(P \wedge \neg T \wedge \neg R)) && \text{Definition B.3.22 (3)}\\ &\equiv (\neg P \vee T \vee R)\ \mathcal{W}\ ((T \vee R) \wedge (\neg P \vee T \vee R)) && \text{Definition B.3.14 (1)}\\ &\equiv (\neg P \vee T \vee R)\ \mathcal{W}\ (R \vee (T \wedge (\neg P \vee T))) && \text{Definition B.3.13 (2)}\\ &\equiv (\neg P \vee T \vee R)\ \mathcal{W}\ (R \vee T) && \text{Definition B.3.12 (1)}\end{aligned}$$

Then we analyze the following statement:

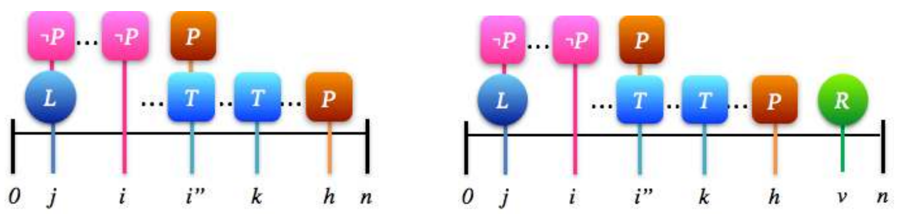
$$(\sigma, j) \vDash (\neg P \vee T \vee R)\ \mathcal{W}\ (R \vee T) \quad \text{iff} \quad \begin{array}{l}(\sigma, j) \vDash (\neg P \vee T \vee R)\ \mathcal{U}\ (R \vee T) \quad \text{or} \\ (\sigma, j) \vDash \square(\neg P \vee T \vee R)\end{array}$$

Therefore, we can analyze two situations:

1. $(\sigma, j) \vDash \square(\neg P \vee T \vee R)$: this formula states that $\neg P$ or T or R holds for all $i, j \le i \le n$ (Definition B.3.18). Here, P can only occur at the same position of T because P cannot hold at the same position of $\neg P$ (Definition B.3.15 (2): contradiction) and the states/events associated with R are not considered in the scope (Section 2.4.1). Thus, the following statements must hold:
   (a) *R does not occur:* in this case, either $\neg P$ or T can hold. If T does not hold neither, then there is no occurrence of P in the defined interval ($[j, n]$). Otherwise, if T holds at some position of the interval, we have that only T occurs, or, T and P hold at the same position.
   (b) *R occurs at position v:* in this case we consider only the events that take place before the occurrence of R, i.e. the interval $[j, (v-1)]$. If no events occur before R, then P does not occur. If T does not hold, then there is no occurrence of P in the defined interval ($[j, n]$). Otherwise, if T holds at some position of the interval, we have that only T occurs, or, T and P hold at the same position.
2. $(\sigma, j) \vDash (\neg P \vee T \vee R)\ \mathcal{U}\ (R \vee T)$: this formula states that $\exists k, j \le k \le n$, s.t. $(\sigma, k) \vDash (R \vee T)$ and $\forall i, j \le i < k$, $(\sigma, i) \vDash (\neg P \vee T \vee R)$ (Definition B.3.19). Here, P can occur at the same position of T or after, because P cannot hold at the same position of $\neg P$ (Definition B.3.15 (2): contradiction) and the states/events associated with R are not considered in the scope (Section 2.4.1). Thus, the following statements must hold:
   (a) *R does not occur (Figure C.6(a)):* in this case we analyze the formula $(\neg P \vee T)\ \mathcal{U}\ T$. There are two possible results for this formula:
      – T holds at $k, j \le k \le n$, $\neg P$ holds $\forall i, j \le i < k$ and P can hold at some position $h, k \le h \le n$.
      – The (last) occurrence of T holds at $k, j \le k \le n$, $\neg P$ and T occur in a non-ordered manner in the interval $[0, (k-1)]$, and P can hold at the same position of T or at some position $h, k \le h \le n$.
   (b) *R occurs at position v (Figure C.6(b)):* in this case we analyze the formula $(\neg P \vee T)\ \mathcal{U}\ (R \vee T)$ because if there is another occurrence of R at some position $m, j \le m < v$ the scope definition would be incomplete since the scope would be limited by the occurrence of R at position m. The formula $(\neg P \vee T)\ \mathcal{U}\ (R \vee T)$ leads us to the following statements:
      – R holds at v and $\neg P$ holds for all $i, j \le i < v$: neither T nor P occurs.
      – T holds at $k, j \le k < n$, $\neg P$ holds $\forall i, j \le i < k$, P can hold at some position $h, k \le h < n$ and R occurs at some position $v, h \le v \le n$ (if $v = h = k$, then neither T nor P occurs; if ($v = h > k$), then T occurs and P does not);
      – R holds at v and T holds $\forall i, j \le i < v$: P may hold at the same position of T.
      – R holds at v and $\neg P$ and T occur in a non-ordered manner in the interval $[j, (v-1)]$: P can take place at the same position of an occurrence of T.

As the analysis results confirm that "T precedes P" holds continuously from the occurrence of L until a future occurrence of R or the end of computation, the safety property is preserved in the interval defined by Q statement. And, since the precedence pattern is represented by $\square Q$ in this scope, we have that $(\sigma, j) \vDash \square Q$ iff $(\sigma, l) \vDash Q$, $\forall l, j \le l \le n$ (Definition B.3.18). Hence, $\square Q$ assures that the safety property represented by Q always occurs in the computation.

■



(a) R does not occur (b) R occurs

Figure C.6: *Precedence pattern - After L Until R scope (Prospec) - Analysis*

C.4 Guarantee Properties: Existence Pattern

The existence pattern states that a given state/event must occur (at least once) within a scope. It is a guarantee property, since in the global scope it corresponds to the canonical formula $\Diamond P$, where P represents the state/event that must hold. Table C.5 presents the LTL formulas for this pattern. The ✓ symbol indicates that other works have already shown that the formula preserves the guarantee property (and therefore it is testable) in the given scope [NGH93, FFJ+10]; the ► symbol indicates the formulas that we analyze in the next sections to show that they also preserve the guarantee property in the given scope; and the $\equiv$ symbol indicates that the formulas of the same row are equivalents.

Table C.5: *LTL formulas for existence pattern [DAC99, SGRE11]*

Pattern	Scope	SPS		Prospec	
Existence (P)	Global	✓	$\Diamond(P)$	$\equiv$	$\Diamond P$
	Before R	►	$\neg R \mathcal{W} (P \wedge \neg R)$	$\equiv$	$\neg((\neg P)\ \mathcal{U}\ R)$
	After L	►	$\Box(\neg L \vee \Diamond(L \wedge \Diamond P))$	►	$\neg((\neg L)\ \mathcal{U}\ (L \wedge \neg\Diamond P))$
	Between L and R	►	$\Box(L \wedge \neg R \rightarrow (\neg R \mathcal{W} (P \wedge \neg R)))$	$\equiv$	$\Box((L \wedge \neg R) \rightarrow (\neg((\neg P)\ \mathcal{U}\ R)))$
	After L Until R	►	$\Box(L \wedge \neg R \rightarrow (\neg R\ \mathcal{U}\ (P \wedge \neg R)))$	$\equiv$	$\Box((L \wedge \neg R) \rightarrow (\neg R\ \mathcal{U}\ (P \wedge \neg R)))$

C.4.1 Before R

Proposition C.4.1 — Prospec - Guarantee Property (Existence Pattern) in the Before R Scope. The *guarantee* property represented by the *existence* pattern is preserved in the *before R* scope.

The existence pattern in the *before R* scope is represented by equivalent formulas in the SPS and the Prospec. Therefore, the proof sketch is the same of Proposition 4.4.1 (Chapter 4). We present below the formulas equivalence.

Equivalence of SPS and Prospec Formulas.

1. SPS Formula $(\neg R\ \mathcal{W}\ (P \wedge \neg R))$ is equivalent to the Prospec Formula from Table C.1 $(\Diamond R \rightarrow \neg R\ \mathcal{U}\ (P \wedge \neg R))$:

$$\begin{aligned} \neg R\ \mathcal{W}\ (P \wedge \neg R) &\equiv (\Box\neg R) \vee (\neg R\ \mathcal{U}\ (P \wedge \neg R)) && \text{Definition B.3.20} \\ &\equiv (\neg\Diamond R) \vee (\neg R\ \mathcal{U}\ (P \wedge \neg R)) && \text{Definition B.3.22 (2)} \\ &\equiv \Diamond R \rightarrow (\neg R\ \mathcal{U}\ (P \wedge \neg R)) && \text{Definition B.3.7} \end{aligned}$$

2. SPS Formula $(\neg R\ \mathcal{W}\ (P \wedge \neg R))$ is equivalent to the Prospec Formula from Table C.1 $(\neg((\neg P)\ \mathcal{U}\ R))$:

$$\neg R\ \mathcal{W}\ (P \wedge \neg R) \equiv \neg((\neg P)\ \mathcal{U}\ R) \qquad \text{Definition B.3.22 (3)}$$

C.4.2 After L

Proposition C.4.2 — Prospec - Guarantee Property (Existence Pattern) in the After L Scope. The *guarantee* property represented by the *existence* pattern is preserved in the *after L* scope.

The existence pattern in the *after L* scope is represented by different formulas in the SPS and the Prospec. Therefore, we present below the proof sketch for the Prospec formula to show the validity of Proposition C.4.2.

Prospec Formula

The analyzed Prospec formula is the formula presented in Table C.1.

$$\neg((\neg L)\ \mathcal{U}\ (L \wedge \neg\Diamond P))$$

Proof Sketch C.4.1. Consider a sequence σ of states $s_0, s_1, \dots s_n$, where s_0 is the initial state and s_n is the final state of the program execution. We have the following equivalence:

$$\begin{array}{rcll}
\neg((\neg L)\ \mathcal{U}\ (L \wedge \neg\Diamond P)) & \equiv & (\neg(L \wedge \neg\Diamond P))\ \mathcal{W}\ (L \wedge (\neg(L \wedge \neg\Diamond P))) & \text{Definition B.3.22 (3)} \\
 & \equiv & (\neg L \vee \Diamond P)\ \mathcal{W}\ (L \wedge (\neg L \vee \Diamond P)) & \text{Definition B.3.14 (1)} \\
 & \equiv & (\neg L \vee \Diamond P)\ \mathcal{W}\ ((L \wedge (\neg L)) \vee (L \wedge (\Diamond P))) & \text{Definition B.3.13 (1)} \\
 & \equiv & (\neg L \vee \Diamond P)\ \mathcal{W}\ \mathsf{False} \vee (L \wedge (\Diamond P))) & \text{Definition B.3.15 (2)} \\
 & \equiv & (\neg L \vee \Diamond P)\ \mathcal{W}\ (L \wedge (\Diamond P)) & \text{Definition B.3.9 (2)}
\end{array}$$

Analyzing the formula $(\neg L \vee \Diamond P)\ \mathcal{W}\ (L \wedge (\Diamond P))$, we have that the following statement must hold, according to Definition B.3.20:

$$(\sigma, 0) \vDash (\neg L \vee \Diamond P)\ \mathcal{W}\ (L \wedge (\Diamond P)) \quad \text{iff} \quad (\sigma, 0) \vDash \Box(\neg L \vee \Diamond P) \quad \text{or} \quad (\neg L \vee \Diamond P)\ \mathcal{U}\ (L \wedge (\Diamond P))$$

Now we must analyze the two cases:

- **Case 1.** The formula $\Box(\neg L \vee \Diamond P)$ states that $\neg L$ holds at every position of σ or that every position in the sequence coincides or is followed by a later position satisfying P (Definitions B.3.5, B.3.18 and B.3.17). In this case, the scope is only defined when at some position k, $0 \leq k \leq n$ we have that $(\sigma, k) \vDash L \wedge \Diamond P$ (i.e. L occurs at k and P holds for some m s.t. $k \leq m \leq n$).

- **Case 2.** The formula $(\neg L \vee \Diamond P)\ \mathcal{U}\ L \wedge (\Diamond P))$ states that either $\neg L$ or $\Diamond P$ holds until $L \wedge (\Diamond P)$ holds (Definition B.3.19). Since the states/events that take place before L can be ignored in this scope definition, we only analyze the formula $(L \wedge (\Diamond P))$ that must hold at some position k, s.t. $0 \leq k \leq n$. This formula states that L occurs at k and P holds for some m s.t. $k \leq m \leq n$.

Considering the two cases, we have that the guarantee property is preserved in *case 2*, and in *case 1*, we have the scope definition for at least one situation. Therefore, we conclude that the guarantee property is preserved in the Prospec formula of *after L* scope.

■

C.4.3 Between L and R

Proposition C.4.3 — Prospec - Guarantee Property (Existence Pattern) in the Between L and R Scope. The *guarantee* property represented by the *existence* pattern is preserved in the *between L and R* scope.

The existence pattern in the *between L and R* scope is represented by equivalent formulas in the SPS and the Prospec. Therefore, the proof sketch is the same of Proposition 4.4.3 (Chapter 4). We present below the formulas equivalence.

Equivalence of SPS and Prospec Formulas. SPS Formula (on the left side) is equivalent to the Prospec Formula from Table C.1 (on the right side):

$$\Box(L \land \neg R \rightarrow (\neg R \mathcal{W}(P \land \neg R))) \quad \equiv \quad \Box((L \land \neg R) \rightarrow (\neg((\neg P)\ \mathcal{U}\ R))) \quad \text{Definition B.3.22 (3)}$$

C.4.4 After L Until R

Proposition C.4.4 — Prospec - Guarantee Property (Existence Pattern) in the After L Until R Scope. The *guarantee* property represented by the *existence* pattern is preserved in the *after L until R* scope.

The existence pattern in the *after L until R* scope is represented by the same formula in the SPS and the Prospec. Therefore, the proof sketch is the same of Proposition 4.4.4 (Chapter 4).

C.5 Recurrence Properties: Response Pattern

The response pattern states that a state/event P must always be followed by a state/event T within a scope. It is a recurrence property, since in the global scope it corresponds to the formula $\Box\Diamond(\neg P\ \mathcal{B}\ T) \equiv \Box(P \rightarrow \Diamond T)$, where T represents the state/event that must respond to the state/event P. Table C.6 presents the LTL formulas for this pattern. The ✓ symbol indicates that other works have already shown that the formula preserves the recurrence property (and therefore it is testable) in the given scope [NGH93, FFJ+10]; the ► symbol indicates the formulas that we analyze in the next sections to show that they also preserve the recurrence property in the given scope; and the ≡ symbol indicates that the formulas of the same row are equivalents.

Table C.6: *LTL formulas for response pattern [DAC99, SGRE11]*

Pattern	Scope	SPS	Prospec
Response (T, P)	Global	✓ $\Box(P \rightarrow \Diamond T)$	≡ $\Box(P \rightarrow \Diamond T)$
	Before R	► $\Diamond R \rightarrow (P \rightarrow (\neg R\ \mathcal{U}\ (T \land \neg R)))\ \mathcal{U}\ R$	► $\neg((\neg R)\ \mathcal{U}\ (P \land (\neg R) \land ((\neg T)\ \mathcal{U}\ R)))$
	After L	► $\Box(L \rightarrow \Box(P \rightarrow \Diamond T))$	► $\neg((\neg L)\ \mathcal{U}\ (L \land (\neg\Box(P \rightarrow \Diamond T))))$
	Between L and R	► $\Box((L \land \neg R \land \Diamond R) \rightarrow (P \rightarrow (\neg R\ \mathcal{U}\ (T \land \neg R)))\ \mathcal{U}\ R)$	► $\Box((L \land \neg R) \rightarrow \neg((\neg R)\ \mathcal{U}\ (P \land (\neg R) \land ((\neg T)\ \mathcal{U}\ R))))$
	After L Until R	► $\Box(L \land \neg R \rightarrow ((P \rightarrow (\neg R\ \mathcal{U}\ (T \land \neg R)))\mathcal{W}R)$	► $\Box((L \land \neg R) \rightarrow \neg((\neg R)\ \mathcal{U}\ (P \land (\neg R) \land ((\Box((\neg T) \land \neg R)) \lor ((\neg T)\ \mathcal{U}\ R)))))$

Before we proceed to the scopes analysis, we give a few recurring definitions in the proof sketches for this pattern.

Definition C.5.1 — $(\neg(P \land (\neg R) \land ((\neg T)\ \mathcal{U}\ R)))$. Consider a sequence σ of states $s_0, s_1, \ldots s_n$, where s_0 is the initial state and s_n is the final state of the program execution, and a position i such that $0 \leq i \leq n$. We first rewrite the statement $(\neg(P \land (\neg R) \land ((\neg T)\ \mathcal{U}\ R)))$ as follows:

$$\begin{aligned}
&(\neg(P \land (\neg R) \land ((\neg T)\ \mathcal{U}\ R))) \\
&\equiv (\neg(P \land (\neg R))) \lor (\neg(\neg T\ \mathcal{U}\ R)) && \text{Definition B.3.14 (1)} \\
&\equiv (\neg P \lor R) \lor (\neg(\neg T\ \mathcal{U}\ R)) && \text{Definition B.3.14 (1)} \\
&\equiv (\neg P \lor R) \lor (\neg R\ \mathcal{W}\ (T \land \neg R)) && \text{Definition B.3.22 (1)}
\end{aligned}$$

Then we proceed to the analysis of the statement:

$$(\sigma, i) \vDash (\neg P \lor R) \lor (\neg R\ \mathcal{W}\ (T \land \neg R))$$

Therefore, one of these three cases must occur (Definition B.3.5):

- **Case 1.** $(\sigma, i) \vDash \neg P$: $\neg P$ holds at position i. P cannot occur, otherwise we would have a contradiction (Definition B.3.15 (2)).
- **Case 2.** $(\sigma, i) \vDash R$: R holds at position i, which means that if there are no later occurrences of L, all states/events that occur at i or after i are not considered in the scope (Section 2.4.1).
- **Case 3.** $(\sigma, i) \vDash (\neg R \; \mathcal{W} \; (T \wedge \neg R))$: here we have two situations to analyze (Definition B.3.20):
  - $(\sigma, i) \vDash \Box \neg R$: R does not hold $\forall h, i \leq h \leq n$ (Definition B.3.18). P and T can also hold here, but we do not have any restriction regarding the order they might occur.
  - $(\sigma, i) \vDash \neg R \; \mathcal{U} \; (T \wedge \neg R)$: there are two statements that must hold (Definition B.3.19):
    * $(\sigma, k) \vDash T \wedge \neg R$: T and $\neg R$ hold at position k, $i \leq k \leq n$. P may occur at k, but R cannot occur because of $\neg R$ ($R \wedge \neg R$ is a contradiction (Definition B.3.15 (2))).
    * $(\sigma, m) \vDash \neg R$: $\forall m, \; i \leq m < k, \neg R$ holds. P may occur in this interval, but T cannot occur because it has already taken place at k (Definition B.3.19). R cannot hold in this interval neither because this would lead to a contradiction, as stated previously.

Hence, we can only consider that T responds to P in *case 3*.

⇒ *The Definition C.5.1 is used in Definition C.5.2, and in Proof-Sketch C.5.4.*

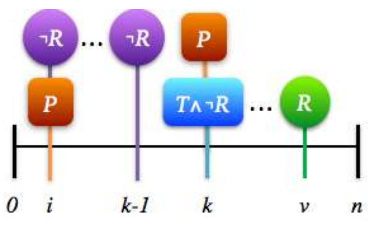


Figure C.7: *Response pattern - Definition C.5.1, Case 3 with a possible occurrence of R*

Definition C.5.2 — $R \wedge (\neg(P \wedge (\neg R) \wedge ((\neg T) \; \mathcal{U} \; R))) \equiv R$. To show the equivalence relation of this definition, we rewrite the statement $R \wedge (\neg(P \wedge (\neg R) \wedge ((\neg T) \; \mathcal{U} \; R)))$ as follows:

$R \wedge (\neg(P \wedge (\neg R) \wedge ((\neg T) \; \mathcal{U} \; R)))$	
$\equiv R \wedge ((\neg P \vee R) \vee (\neg R \; \mathcal{W} \; (T \wedge \neg R)))$	Definition C.5.1
$\equiv (R \wedge (\neg P \vee R)) \vee (R \wedge (\neg R \; \mathcal{W} \; (T \wedge \neg R)))$	Definition B.3.13 (1)
$\equiv R \vee (R \wedge (\neg R \; \mathcal{W} \; (T \wedge \neg R)))$	Definition B.3.12 (1)
$\equiv R$	Definition B.3.12 (2)

⇒ *The Definition C.5.2 is used in Definition C.5.3.*

Definition C.5.3 — $\neg((\neg R) \; \mathcal{U} \; (P \wedge (\neg R) \wedge ((\neg T) \; \mathcal{U} \; R)))$. Consider a sequence σ of states $s_0, s_1, \ldots s_n$, where s_0 is the initial state and s_n is the final state of the program execution, and a position i

such that $0 \leq i \leq n$. We rewrite the formula $\neg((\neg R)\ \mathcal{U}\ (P \wedge (\neg R) \wedge ((\neg T)\ \mathcal{U}\ R)))$ as follows:

$$\begin{array}{ll} \neg((\neg R)\ \mathcal{U}\ (P \wedge (\neg R) \wedge ((\neg T)\ \mathcal{U}\ R))) & \\ \equiv (\neg(P \wedge (\neg R) \wedge ((\neg T)\ \mathcal{U}\ R)))\ \mathcal{W}\ R \wedge (\neg(P \wedge (\neg R) \wedge ((\neg T)\ \mathcal{U}\ R))) & \text{Definition B.3.22 (3)} \\ \equiv (\neg(P \wedge (\neg R) \wedge ((\neg T)\ \mathcal{U}\ R)))\ \mathcal{W}\ R & \text{Definition C.5.2} \\ \equiv ((\neg P \vee R) \vee (\neg R\ \mathcal{W}\ (T \wedge \neg R)))\ \mathcal{W}\ R & \text{Definition C.5.1} \end{array}$$

Now, analyzing the formula $((\neg P \vee R) \vee (\neg R\ \mathcal{W}\ (T \wedge \neg R)))\ \mathcal{W}\ R$ (Definition B.3.20):

- **Case 1.** $(\sigma, m) \models \Box((\neg P \vee R) \vee (\neg R\ \mathcal{W}\ (T \wedge \neg R)))$: the statements of Definition C.5.1 hold for all m, s.t. $i \leq m \leq n$ (Definition B.3.18). Here, if we want to assure that T responds to P then the statements of Definition C.5.1 - *Case 3* must hold at position i or after the occurrence of $\neg P$. R may occur at some position after T, to assure that T responds to P before R (Figure C.7).

- **Case 2.** $(\sigma, i) \models ((\neg P \vee R) \vee (\neg R\ \mathcal{W}\ (T \wedge \neg R)))\ \mathcal{U}\ R$: according to Definitions B.3.19 and C.5.1, we have that the following statements must hold:

  - $(\sigma, v) \models R$: R holds at position v, $i < v \leq n$.
  - $(\sigma, l) \models ((\neg P \vee R) \vee (\neg R\ \mathcal{W}\ (T \wedge \neg R)))$: $((\neg P \vee R) \vee (\neg R\ \mathcal{W}\ (T \wedge \neg R)))$ holds for all l, s.t. $i \leq l < v$. Here, if we want to assure that T responds to P before the occurrence of R then the statements of Definition C.5.1 - *Case 3* must hold (Figure C.7).

$\Rightarrow$ *The Definition C.5.3 is used in Proof-Sketches C.5.1 and C.5.3.*

Remark. According to Manna and Pnueli [MP92], the following formulas represent the *response* pattern (recurrence property):

- $\Box\Diamond((\neg P)\ \mathcal{B}\ T)$: this formula has the canonical form of a recurrence property (Section 3.2.2). It states that there are (infinitely) many positions in which all previous requests have been responded to. These are positions such that no new requests (represented by P) have been posted since the last response (represented by T) (Definitions B.3.18, B.3.24 and B.3.27)[MP92].

- $\Box(P \rightarrow \Diamond T)$: this formula keeps the semantics of a recurrence property. It states that for each position j, $0 \leq j \leq n$, if P holds at j there must be a later position k, $j \leq k \leq n$ satisfying T (Definitions B.3.18 and B.3.17). In other words, every position in which P holds coincides or is followed by a position in which T holds [MP92].

Therefore, we have that $\Box\Diamond((\neg P)\ \mathcal{B}\ T) \equiv \Box(P \rightarrow \Diamond T)$.

C.5.1 Before R

Proposition C.5.1 — Prospec - Recurrence Property (Response Pattern) in the Before R Scope. The *recurrence* property represented by the *response* pattern is preserved in the *before R* scope.

The response pattern in the *before R* scope is represented by different formulas in the SPS and the Prospec. Therefore, we present below the proof sketch for the Prospec formula to show the validity of Proposition C.5.1.

Prospec Formula

The analyzed Prospec formula is the formula presented in Table C.1.

$$\neg((\neg R)\ \mathcal{U}\ (P \wedge (\neg R) \wedge ((\neg T)\ \mathcal{U}\ R)))$$

Proof Sketch C.5.1. Consider a sequence σ of states $s_0, s_1, \ldots s_n$, where s_0 is the initial state and s_n is the final state of the program execution. The following statement must hold:

$$(\sigma, 0) \vDash \neg((\neg R)\ \mathcal{U}\ (P \wedge (\neg R) \wedge ((\neg T)\ \mathcal{U}\ R)))$$

This statement is analyzed in Definition C.5.3. Considering in this definition that $i = 0$ and that R occurs at least once after T responds to P, we assure that the recurrence property is preserved in this scope (Figure C.7).

■

C.5.2 After L

Proposition C.5.2 — Prospec - Recurrence Property (Response Pattern) in the After L Scope. The *recurrence* property represented by the *response* pattern is preserved in the *after L* scope.

The response pattern in the *after L* scope is represented by different formulas in the SPS and the Prospec. Therefore, we present below the proof sketch for the Prospec formula to show the validity of Proposition C.5.2.

Prospec Formula

The analyzed Prospec formula is the formula presented in Table C.1.

$$\neg((\neg L)\ \mathcal{U}\ (L \wedge (\neg\Box(P \rightarrow \Diamond T))))$$

Proof Sketch C.5.2. Consider a sequence σ of states $s_0, s_1, \ldots s_n$, where s_0 is the initial state and s_n is the final state of the program execution. We have the following equivalence:

$\neg((\neg L)\ \mathcal{U}\ (L \wedge (\neg\Box(P \rightarrow \Diamond T))))$	
$\equiv (\neg(L \wedge \neg\Box(P \rightarrow \Diamond T)))\ \mathcal{W}\ (L \wedge (\neg(L \wedge \neg\Box(P \rightarrow \Diamond T))))$	Definition B.3.22 (3)
$\equiv (\neg L \vee \Box(P \rightarrow \Diamond T))\ \mathcal{W}\ (L \wedge (\neg L \vee \Box(P \rightarrow \Diamond T)))$	Definition B.3.14 (1)
$\equiv (\neg L \vee \Box(P \rightarrow \Diamond T))\ \mathcal{W}\ ((L \wedge (\neg L)) \vee (L \wedge \Box(P \rightarrow \Diamond T)))$	Definition B.3.13 (1)
$\equiv (\neg L \vee \Box(P \rightarrow \Diamond T))\ \mathcal{W}\ \mathit{False} \vee (L \wedge \Box(P \rightarrow \Diamond T)))$	Definition B.3.15 (2)
$\equiv (\neg L \vee \Box(P \rightarrow \Diamond T))\ \mathcal{W}\ (L \wedge \Box(P \rightarrow \Diamond T))$	Definition B.3.9 (2)

Thus, we can analyze the following statement:

$$\begin{aligned}&(\neg L \vee \Box(P \rightarrow \Diamond T))\ \mathcal{W}\ (L \wedge \Box(P \rightarrow \Diamond T))\\ &\qquad\text{iff}\quad \Box(\neg L \vee \Box(P \rightarrow \Diamond T))\\ &\qquad\text{or}\quad (\neg L \vee \Box(P \rightarrow \Diamond T))\ \mathcal{U}\ (L \wedge \Box(P \rightarrow \Diamond T))\end{aligned}$$

Analyzing the two cases:

- **Case 1.** The formula $\Box(\neg L \vee \Box(P \rightarrow \Diamond T))$ states that either $\neg L$ holds at every position of σ or $\Box(P \rightarrow \Diamond T)$ holds at every position of σ (Definitions B.3.5 and B.3.18). In this case, the scope is only defined when at some position k, $0 \leq k \leq n$ we have that $(\sigma, k) \vDash L \wedge \Box(P \rightarrow \Diamond T)$ (i.e. L occurs at k and $\Box(P \rightarrow \Diamond T)$, which is the response pattern in the global scope (Tables 2.3 and C.1), holds at k).
- **Case 2.** The formula $(\neg L \vee \Box(P \rightarrow \Diamond T))\ \mathcal{U}\ (L \wedge \Box(P \rightarrow \Diamond T))$ states that either $\neg L$ or $\Box(P \rightarrow \Diamond T)$ holds until $L \wedge \Box(P \rightarrow \Diamond T)$ holds. (Definition B.3.19). Since the states/events that take place before L can be ignored in this scope definition (Section 2.4.1), we only analyze the formula $(L \wedge \Box(P \rightarrow \Diamond T))$ that must hold at some position k, s.t. $0 \leq k \leq n$. Then, considering the execution at position k, we have that the following statements must hold:

- The formula L holds at k, $0 \leq k \leq n$;
- The formula $\square(P \rightarrow \Diamond T)$, which is the response pattern in the global scope (Tables 2.3 and C.1), holds at k, $0 \leq k \leq n$. This means that P occurs at some position i, $k \leq i \leq n$ and T occurs at some position j, $i \leq j \leq n$. i.e., T responds to P.

Considering the two cases, we have that the recurrence property is preserved in *case 2*, and in *case 1*, we have the scope definition in at least one situation. Therefore, we conclude that the recurrence property is preserved in the Prospec formula of *after L* scope. ■

C.5.3 Between L and R

Proposition C.5.3 — Prospec - Recurrence Property (Response Pattern) in the Between L and R Scope. The *recurrence* property represented by the *response* pattern is preserved in the *between L and R* scope.

The response pattern in the *between L and R* scope is represented by different formulas in the SPS and the Prospec. Therefore, we present below the proof sketch for the Prospec formula to show the validity of Proposition C.5.3.

Prospec Formula

The analyzed Prospec formula is the formula presented in Table C.1.

$$\square Q, \text{ where } Q = ((L \wedge \neg R) \rightarrow \neg((\neg R)\ \mathcal{U}\ (P \wedge (\neg R) \wedge ((\neg T)\ \mathcal{U}\ R))))$$

Proof Sketch C.5.3. Consider a sequence σ of states $s_0, s_1, \dots s_n$, where s_0 is the initial state and s_n is the final state of the program execution, and a position j such that $0 \leq j \leq n$. We first analyze the Q statement, using Definition B.3.7:

$$\begin{aligned}(\sigma, j) \vDash ((L \wedge \neg R) \rightarrow \neg((\neg R)\ \mathcal{U}\ (P \wedge (\neg R) \quad \wedge((\neg T)\ \mathcal{U}\ R)))) \quad \text{therefore} \quad &\text{if } (\sigma, j) \vDash (L \wedge \neg R)\\ &\text{then } (\sigma, j) \vDash \neg((\neg R)\ \mathcal{U}\ (P \wedge (\neg R) \wedge ((\neg T)\ \mathcal{U}\ R)))\end{aligned}$$

Using this statement, we can analyze each part of the implication as follows:

- **Case 1.** $(\sigma, j) \vDash L \wedge \neg R$ iff $(\sigma, j) \vDash L$ and $(\sigma, j) \vDash \neg R$. This formula deals with the states/events that limit the scope, and says that L occurs at position j and R does not.
- **Case 2.** $(\sigma, j) \vDash \neg((\neg R)\ \mathcal{U}\ (P \wedge (\neg R) \wedge ((\neg T)\ \mathcal{U}\ R)))$. It is the Definition C.5.3, considering that $i = j$ and R occurs at least once after T responds to P.

Analyzing both parts of the implication, we have that the first part guarantees that L occurs at j, and, the second part assures that T responds to P before the occurrence of R at some position ν, $j < \nu \leq n$ (Figure C.7). Hence, we assure that T always responds to P, and the recurrence property is preserved in the Q statement. In addition, since the response pattern is represented by $\square Q$ in this scope, we have that $(\sigma, j) \vDash \square Q$ iff $(\sigma, l) \vDash Q$, $\forall l,\ j \leq l \leq n$ (Definition B.3.18). Hence, $\square Q$ assures that the recurrence property represented by Q always occurs in the computation (Figure C.8). ■

C.5.4 After L Until R

Proposition C.5.4 — Prospec - Recurrence Property (Response Pattern) in the After L Until R Scope. The *recurrence* property represented by the *response* pattern is preserved in the *after L until R* scope.

The response pattern in the *after L until R* scope is represented by different formulas in the SPS and the Prospec. Therefore, we present below the proof sketch for the Prospec formula to show the validity of Proposition C.5.4.

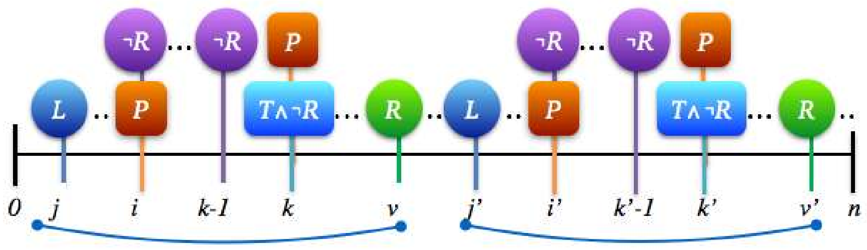


Figure C.8: *Response pattern - Between L and R scope (Prospec) - Scope repetition*

Prospec Formula

The analyzed Prospec formula is the formula presented in Table C.1.

$$\Box Q, \text{ where } Q = ((L \wedge \neg R) \rightarrow \neg((\neg R)\ \mathcal{U}\ (P \wedge (\neg R) \wedge ((\Box((\neg T) \wedge \neg R)) \vee ((\neg T)\ \mathcal{U}\ R)))))$$

Proof Sketch C.5.4. Consider a sequence σ of states $s_0, s_1, \ldots s_n$, where s_0 is the initial state and s_n is the final state of the program execution, and a position j such that $0 \le j \le n$. We first analyze the Q statement using Definition B.3.7:

$$\begin{aligned}(\sigma, j) \vDash (L \wedge \neg R) \rightarrow\ & \neg((\neg R)\ \mathcal{U}\ (P \wedge (\neg R) \wedge ((\Box((\neg T) \wedge \neg R)) \vee ((\neg T)\ \mathcal{U}\ R))))\\ & \text{therefore} \quad \text{if } (\sigma, j) \vDash (L \wedge \neg R)\\ & \text{then } (\sigma, j) \vDash \neg((\neg R)\ \mathcal{U}\ (P \wedge (\neg R) \wedge ((\Box((\neg T) \wedge \neg R)) \vee ((\neg T)\ \mathcal{U}\ R))))\end{aligned}$$

The first part of the implication states that L holds at position j and R cannot hold at this same position. The second part of the implication requires a more detailed analysis. In this analysis, we use the following definitions:

Definition C.5.4 — $(\neg(P \wedge (\neg R) \wedge ((\Box(\neg T \wedge \neg R)) \vee ((\neg T)\ \mathcal{U}\ R))))$**.** Consider a position i such that $0 \le i \le n$. We first rewrite the statement $(\neg(P \wedge (\neg R) \wedge ((\Box(\neg T \wedge \neg R)) \vee ((\neg T)\ \mathcal{U}\ R))))$ as follows:

$$\begin{aligned}& (\neg(P \wedge (\neg R) \wedge ((\Box(\neg T \wedge \neg R)) \vee ((\neg T)\ \mathcal{U}\ R))))\\ \equiv\ & (\neg(P \wedge (\neg R))) \vee (\neg((\Box(\neg T \wedge \neg R)) \vee ((\neg T)\ \mathcal{U}\ R))) && \text{Definition B.3.14 (1)}\\ \equiv\ & (\neg P \vee R) \vee (\neg((\Box(\neg T \wedge \neg R)) \vee ((\neg T)\ \mathcal{U}\ R))) && \text{Definition B.3.14 (1)}\\ \equiv\ & (\neg P \vee R) \vee (\neg(\Box(\neg T \wedge \neg R)) \wedge \neg((\neg T)\ \mathcal{U}\ R)) && \text{Definition B.3.14 (2)}\\ \equiv\ & (\neg P \vee R) \vee (\Diamond(\neg(\neg T \wedge \neg R)) \wedge \neg((\neg T)\ \mathcal{U}\ R)) && \text{Definition B.3.22 (1)}\\ \equiv\ & (\neg P \vee R) \vee ((\Diamond(T \vee R)) \wedge \neg((\neg T)\ \mathcal{U}\ R)) && \text{Definition B.3.14 (1)}\\ \equiv\ & (\neg P \vee R) \vee ((\Diamond(T \vee R)) \wedge (\neg R\ \mathcal{W}\ (T \wedge \neg R))) && \text{Definition B.3.22 (3)}\end{aligned}$$

We proceed to the analysis of the formula:

$$(\neg P \vee R) \vee ((\Diamond(T \vee R)) \wedge (\neg R\ \mathcal{W}\ (T \wedge \neg R)))$$

It is similar to the results of Definition C.5.1 with the additional information that when $(\neg R\ \mathcal{W}\ (T \wedge \neg R))$ holds then either T or R occurs in the future. Thus, we have that one of these cases must hold:

- **Case 1.** $(\sigma, i) \vDash \neg P$: $\neg P$ holds at position i. P cannot occur, otherwise we would have a contradiction (Definition B.3.15 (2)).
- **Case 2.** $(\sigma, i) \vDash R$: R holds at position i, which means that if there are no later occurrences of L, all states/events that occur at i or after i are not considered in the scope (Section 2.4.1).

- **Case 3.** $(\sigma, i) \models ((\Diamond(T \vee R)) \wedge (\neg R\ \mathcal{W}\ (T \wedge \neg R)))$: here we have two situations to analyze (Definition B.3.20):
  - $(\sigma, i) \models (\Diamond(T \vee R)) \wedge \Box\neg R$: R does not hold $\forall h, i \leq h \leq n$ (Definition B.3.18). T holds at some position between i and n, but we do not have any information regarding the occurrence of P.
  - $(\sigma, i) \models (\Diamond(T \vee R)) \wedge \neg R\ \mathcal{U}\ (T \wedge \neg R)$: there are two statements that must hold (Definition B.3.19) when T occurs and R *does not occur in the computation*:
    - $(\sigma, k) \models T \wedge \neg R$: T and $\neg R$ hold at position k, $i \leq k \leq n$. P may occur at k, but R cannot occur because of $\neg R$ ($R \wedge \neg R$ is a contradiction (Definition B.3.15 (2))).
    - $(\sigma, m) \models \neg R$: $\forall m,\ i \leq m < k$, $\neg R$ holds. P may occur in this interval, but T cannot occur because it has already taken place at k (Definition B.3.19). R cannot hold in this interval neither because this would lead to a contradiction, as stated previously.

    If R *occurs in the computation* then we have three statements that must hold:
    - $(\sigma, k) \models T \wedge \neg R$: T and $\neg R$ hold at position k, $i \leq k < n$. P may occur at k, but R cannot occur because of $\neg R$ ($R \wedge \neg R$ is a contradiction (Definition B.3.15 (2))).
    - $(\sigma, m) \models \neg R$: $\forall m,\ i \leq m < k$, $\neg R$ holds. P may occur in this interval, but T cannot occur because it has already taken place at k (Definition B.3.19). R cannot hold in this interval neither because this would lead to a contradiction, as stated previously.
    - $(\sigma, v) \models R$: R occurs at some position v, s.t. $k < v \leq n$.

Hence, we can only consider that T responds to P in *case 3*.

Definition C.5.5 — $R \wedge (\neg(P \wedge (\neg R) \wedge ((\Box(\neg T \wedge \neg R)) \vee ((\neg T)\ \mathcal{U}\ R)))) \equiv R$. We show the equivalence as follows:

$$\begin{array}{ll}
R \wedge (\neg(P \wedge (\neg R) \wedge ((\Box(\neg T \wedge \neg R)) \vee ((\neg T)\ \mathcal{U}\ R)))) & \\
\equiv R \wedge ((\neg P \vee R) \vee ((\Diamond(T \vee R)) \wedge (\neg R\ \mathcal{W}\ (T \wedge \neg R)))) & \text{Definition C.5.4} \\
\equiv (R \wedge (\neg P \vee R)) \vee (R \wedge ((\Diamond(T \vee R)) \wedge (\neg R\ \mathcal{W}\ (T \wedge \neg R)))) & \text{Definition B.3.13 (1)} \\
\equiv R \vee (R \wedge ((\Diamond(T \vee R)) \wedge (\neg R\ \mathcal{W}\ (T \wedge \neg R)))) & \text{Definition B.3.12 (1)} \\
\equiv R & \text{Definition B.3.12 (1)}
\end{array}$$

Using the above definitions, we can rewrite the second part of the implication as follows:

$$\begin{array}{ll}
\neg((\neg R)\ \mathcal{U}\ (P \wedge (\neg R) \wedge ((\Box((\neg T) \wedge \neg R)) \vee ((\neg T)\ \mathcal{U}\ R)))) & \\
\equiv (\neg(P \wedge (\neg R) \wedge ((\Box((\neg T) \wedge \neg R)) \vee ((\neg T)\ \mathcal{U}\ R))))\ \mathcal{W}\ (R \wedge & \\
\quad (\neg(P \wedge (\neg R) \wedge ((\Box((\neg T) \wedge \neg R)) \vee ((\neg T)\ \mathcal{U}\ R))))) & \text{Definition B.3.22 (3)} \\
\equiv (\neg(P \wedge (\neg R) \wedge ((\Box((\neg T) \wedge \neg R)) \vee ((\neg T)\ \mathcal{U}\ R))))\ \mathcal{W}\ R & \text{Definition C.5.5} \\
\equiv ((\neg P \vee R) \vee ((\Diamond(T \vee R)) \wedge (\neg R\ \mathcal{W}\ (T \wedge \neg R))))\ \mathcal{W}\ R & \text{Definition C.5.4}
\end{array}$$

And combining the two possible situations of the formula $((\neg P \vee R) \vee ((\Diamond(T \vee R)) \wedge (\neg R\ \mathcal{W}\ (T \wedge \neg R))))\ \mathcal{W}\ R$ (Definition B.3.20) with the first part of the implication, we have that the following statements must hold:

- $(\sigma, m) \models \Box((\neg P \vee R) \vee ((\Diamond(T \vee R)) \wedge (\neg R\ \mathcal{W}\ (T \wedge \neg R))))$: the *cases 1 and 3* of Definition C.5.4 hold for all m, s.t. $j \leq m \leq n$ (Definition B.3.18). Here, if we want to assure that T responds to P then the statements of Definition C.5.4 - *Case 3* must hold at position j or after the occurrence of $\neg P$. R *may or may not occur at some position after* T, to assure that T responds to P before R (Figure C.7).
- $(\sigma, j) \models ((\neg P \vee R) \vee ((\Diamond(T \vee R)) \wedge (\neg R\ \mathcal{W}\ (T \wedge \neg R))))\ \mathcal{U}\ R$: according to Definitions B.3.19 and C.5.4, we have that the following statements must hold:

  - $(\sigma, \nu) \vDash R$: R holds at position ν, $j < \nu \leq n$. Here, R *definitely occurs in the computation*.
  - $(\sigma, l) \vDash ((\neg P \vee R) \vee ((\Diamond(T \vee R)) \wedge (\neg R \ \mathcal{W} \ (T \wedge \neg R))))$: this formula holds for all l, s.t. $j \leq l < \nu$. Here, if we want to assure that T responds to P before the occurrence of R then the statements of Definition C.5.4 - *Case 3* must hold.

- $(\sigma, j) \vDash (L \wedge \neg R)$: L holds at position j, and R does not hold at this position.

Therefore, we have that Q is a recurrence property in the scope *after L until R*. And, since the response pattern is represented by $\Box Q$ in this scope, we have that $(\sigma, j) \vDash \Box Q$ iff $(\sigma, l) \vDash Q$,$\forall l$, $j \leq l \leq n$ (Definition B.3.18). Hence, $\Box Q$ assures that the recurrence property represented by Q always occurs in the computation (Figure C.9).

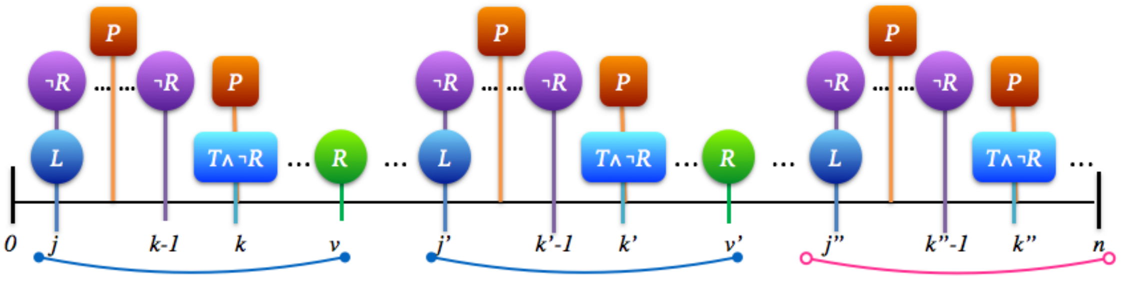


Figure C.9: *Response pattern - After L Until R scope (Prospec) - Scope repetition*

■

C.6 Results

Table C.7 presents all the SPS and Prospec formulas that can be used to specify testable properties. The ✓ symbol indicates that other works have already shown that the formula preserves the property (and therefore it is testable) in the given scope [NGH93, FFJ+10]; the ► symbol indicates the formulas that we analyzed to show that they also preserve the semantics of their respective property class/pattern; and, the ≡ symbol indicates that the formulas of the same row are equivalents.

As we can see from Table C.7, the results we obtained with this study are similar to the ones we obtained in Chapter 4. Therefore, the next theorems are valid for the Prospec formulas.

Theorem C.6.1 — Safety Property in the SPS Scopes - Prospec. The *safety* property represented by the *universality*, *absence* and *precedence* patterns is preserved in all SPS scopes.

The result for the global scope can be reached directly [DAC99, MP92]. The results for the other scopes are given by Propositions C.1.1, C.1.2, C.1.3 and C.1.4 for the universality pattern; Propositions C.2.1, C.2.2, C.2.3 and C.2.4 for the absence pattern; and, Propositions C.3.1, C.3.2, C.3.3 and C.3.4 for the precedence pattern.

Theorem C.6.2 — Guarantee Property in the SPS Scopes - Prospec. The *guarantee* property represented by the *existence* pattern is preserved in all SPS scopes.

The result for the global scope can be reached directly [DAC99, MP92], and the results for the other scopes are given by Propositions C.4.1, C.4.2, C.4.3 and C.4.4.

Theorem C.6.3 — Recurrence Property in the SPS Scopes - Prospec. The *recurrence* property represented by the *response* pattern is preserved in all SPS scopes.

The result for the global scope can be reached directly [DAC99, MP92], and the results for the other scopes are given by Propositions C.5.1, C.5.2, C.5.3 and C.5.4.

Table C.7: *Testable LTL formulas (SPS and Prospec) for Patterns and Scopes [DAC99, SGRE11]*

Pattern	Scope	SPS	Prospec
Universality (P)	Global	✓ $\Box(P)$	$\equiv$ $\Box P$
	Before R	► $\Diamond R \rightarrow (P\ \mathcal{U}\ R)$	$\equiv$ $\Diamond R \rightarrow (P\ \mathcal{U}\ R)$
	After L	► $\Box(L \rightarrow \Box(P))$	► $\neg L\ \mathcal{W}\ (L \wedge \Box P))$
	Between L and R	► $\Box((L \wedge \neg R \wedge \Diamond R) \rightarrow (P\ \mathcal{U}\ R))$	$\equiv$ $\Box((L \wedge \neg R \wedge \Diamond R) \rightarrow (P\ \mathcal{U}\ R))$
	After L Until R	► $\Box(L \wedge \neg R \rightarrow (P \mathcal{W} R))$	► $\Box(L \wedge \neg R \rightarrow (P\ \mathcal{W}\ R))$
Absence (P)	Global	✓ $\Box(\neg P)$	$\equiv$ $\neg(\Diamond P)$
	Before R	► $\Diamond R \rightarrow (\neg P\ \mathcal{U}\ R)$	► $\Diamond R \rightarrow \neg(\neg R\ \mathcal{U}\ P)$
	After L	► $\Box(L \rightarrow \Box(\neg P))$	► $\neg((\neg L)\ \mathcal{U}\ (L \wedge \Diamond P))$
	Between L and R	► $\Box((L \wedge \neg R \wedge \Diamond R) \rightarrow (\neg P\ \mathcal{U}\ R))$	► $\Box((L \wedge \neg R \wedge \Diamond R) \rightarrow \neg(\neg R\ \mathcal{U}\ P))$
	After L Until R	► $\Box(L \wedge \neg R \rightarrow (\neg P \mathcal{W} R))$	► $\Box(L \wedge \neg R \rightarrow \neg(\neg R\ \mathcal{U}\ P))$
Precedence (T, P)	Global	✓ $\neg P \mathcal{W} T$	$\equiv$ $\neg((\neg T)\ \mathcal{U}\ (P \wedge \neg T))$
	Before R	► $\Diamond R \rightarrow (\neg P\ \mathcal{U}\ (T \vee R))$	$\equiv$ $\Diamond R \rightarrow (\neg P\ \mathcal{U}\ (T \vee R))$
	After L	► $\Box \neg L \vee \Diamond(L \wedge (\neg P \mathcal{W} T))$	► $\neg((\neg L)\ \mathcal{U}\ (L \wedge ((\neg T)\ \mathcal{U}\ (P \wedge \neg T))))$
	Between L and R	► $\Box((L \wedge \neg R \wedge \Diamond R) \rightarrow$ $(\neg P\ \mathcal{U}\ (T \vee R)))$	$\equiv$ $\Box((L \wedge \neg R \wedge \Diamond R) \rightarrow (\neg P\ \mathcal{U}\ (T \vee R)))$
	After L Until R	► $\Box(L \wedge \neg R \rightarrow (\neg P \mathcal{W}(T \vee R)))$	► $\Box((L \wedge \neg R) \rightarrow$ $(\neg(((\neg T) \wedge \neg R)\ \mathcal{U}\ (P \wedge (\neg T) \wedge \neg R))))$
Existence (P)	Global	✓ $\Diamond(P)$	$\equiv$ $\Diamond P$
	Before R	► $\neg R \mathcal{W}(P \wedge \neg R)$	$\equiv$ $\neg((\neg P)\ \mathcal{U}\ R)$
	After L	► $\Box(\neg L \vee \Diamond(L \wedge \Diamond P))$	► $\neg((\neg L)\ \mathcal{U}\ (L \wedge \neg \Diamond P))$
	Between L and R	► $\Box(L \wedge \neg R \rightarrow (\neg R \mathcal{W}(P \wedge \neg R)))$	$\equiv$ $\Box((L \wedge \neg R) \rightarrow (\neg((\neg P)\ \mathcal{U}\ R)))$
	After L Until R	► $\Box(L \wedge \neg R \rightarrow (\neg R\ \mathcal{U}\ (P \wedge \neg R)))$	$\equiv$ $\Box((L \wedge \neg R) \rightarrow (\neg R\ \mathcal{U}\ (P \wedge \neg R)))$
Response (T, P)	Global	✓ $\Box(P \rightarrow \Diamond T)$	$\equiv$ $\Box(P \rightarrow \Diamond T)$
	Before R	► $\Diamond R \rightarrow (P \rightarrow (\neg R\ \mathcal{U}\ (T \wedge \neg R)))\ \mathcal{U}\ R$	► $\neg((\neg R)\ \mathcal{U}\ (P \wedge (\neg R) \wedge ((\neg T)\ \mathcal{U}\ R)))$
	After L	► $\Box(L \rightarrow \Box(P \rightarrow \Diamond T))$	► $\neg((\neg L)\ \mathcal{U}\ (L \wedge (\neg \Box(P \rightarrow \Diamond T))))$
	Between L and R	► $\Box((L \wedge \neg R \wedge \Diamond R) \rightarrow$ $(P \rightarrow (\neg R\ \mathcal{U}\ (T \wedge \neg R)))\ \mathcal{U}\ R)$	► $\Box((L \wedge \neg R) \rightarrow$ $\neg((\neg R)\ \mathcal{U}\ (P \wedge (\neg R) \wedge ((\neg T)\ \mathcal{U}\ R))))$
	After L Until R	► $\Box(L \wedge \neg R \rightarrow$ $((P \rightarrow (\neg R\ \mathcal{U}\ (T \wedge \neg R)))\mathcal{W} R)$	► $\Box((L \wedge \neg R) \rightarrow$ $\neg((\neg R)\ \mathcal{U}\ (P \wedge (\neg R) \wedge$ $((\Box((\neg T) \wedge \neg R)) \vee ((\neg T)\ \mathcal{U}\ R)))))$

D — SPS Automata Library

In this appendix, we present the automata library we generated for use in the JPF-SPS tool (Chapter 6). The step-by-step guide for the generation of automata from SPS regular expressions is described in details in Section 5.1.2. Here, we present a summarized version of the process:

SPS Automata Generation:

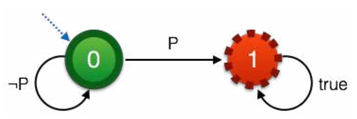
1. Generate the minimal DFA for each SPS regular expression (Table 2.4) that can provide a testable property specification using JFLAP [JFL] (Section 5.1.1). This DFA describes the paths that lead to accepting states;
2. Complete the DFA with inputs that may be missing;
3. Build the complement of the DFA with completed inputs, to define the paths that lead to rejecting states;
4. Combine the DFA and its complement, to generate the property DFA with accepting and rejecting states.

To represent the DFA graphically, we used circles for states and directed arrows for edges. The accepting states are represented by circles with thick borders; the rejecting states are represented by circles with dashed borders; and regular states are represented by circles with no border. Each state is identified by a number, and each edge has a label (event name) to indicate the possible execution paths that lead to either an accepting or a rejecting state.

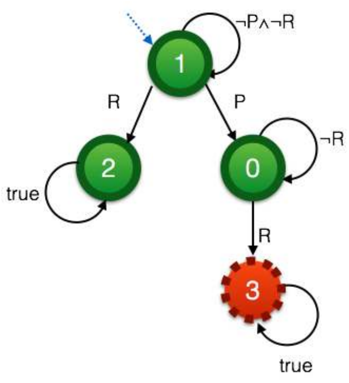
The next sections present the generated automata collection for each SPS pattern that can provide a testable property specification (*absence*, *universality*, *precedence*, *existence* and *response*) in different SPS scopes (*global*, *before R*, *after L*, *between L and R*, and *after L until R*). For a description of SPS patterns and scopes, please refer to Section 2.4.

D.1 Absence Pattern

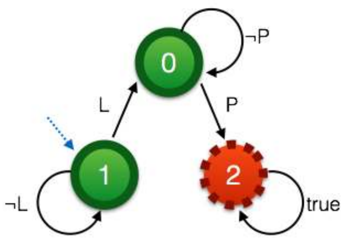
The *absence* pattern states that a given event P does not occur within a scope [DAC99]. The automata that represent this pattern through the SPS scopes (*global*, *before R*, *after L*, *between L and R*, and *after L until R*) are presented below.



(a) *Global* scope

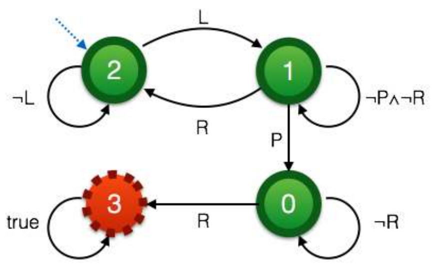


(b) *Before R* scope

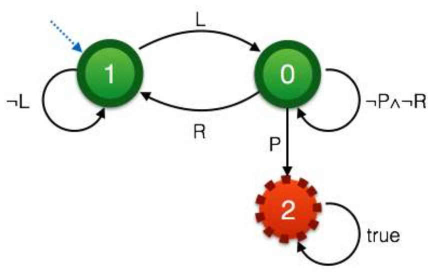


(c) *After L* scope

Figure D.1: *Absence Pattern - DFA Collection*



(a) *Between L and R* scope



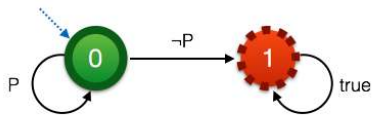
(b) *After L until R* scope

Figure D.2: *Absence Pattern - DFA Collection (cont.)*

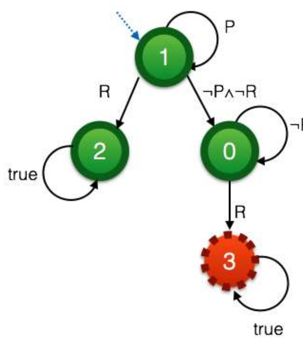
D.2 Universality Pattern

The *universality* pattern states that a given event P occurs throughout a scope [DAC99]. The automata that represent this pattern through the SPS scopes (*global*, *before R*, *after L*, *between L and R*, and *after L until R*) are presented next.

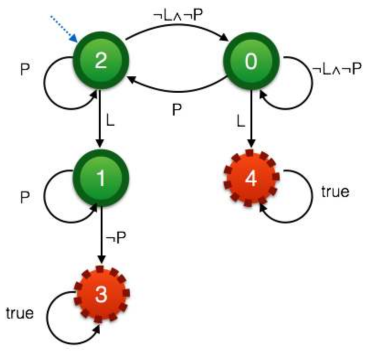
Remark. In the regular expression representation of SPS, the authors used the letter N to represent the negative event. Without loss of generality, we considered in our automata generation that N ≡ [−P]. Thus, the label N does not appear in the generated automata.



(a) *Global* scope

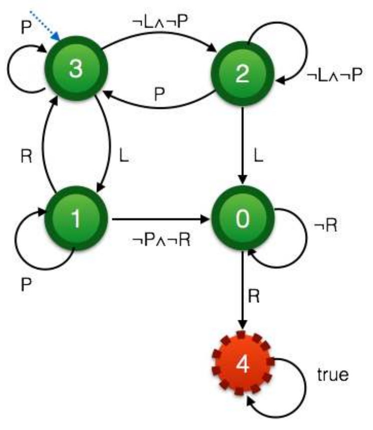


(b) *Before R* scope

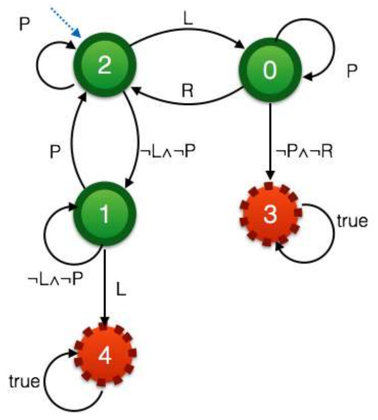


(c) *After L* scope

Figure D.3: *Universality Pattern - DFA Collection*



(a) *Between L and R* scope

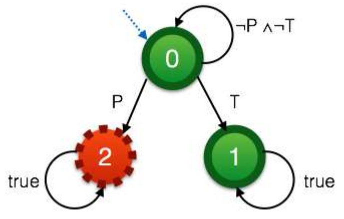


(b) *After L until R* scope

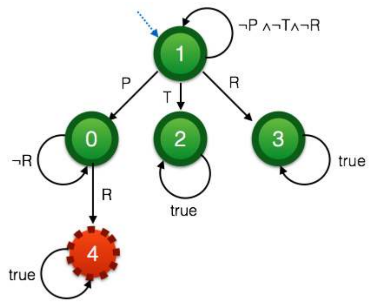
Figure D.4: *Universality Pattern - DFA Collection (cont.)*

D.3 Precedence Pattern

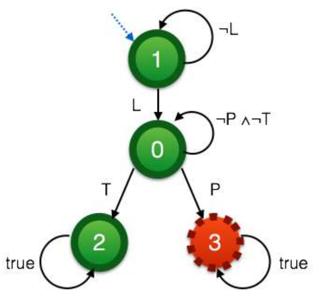
The *precedence* pattern says that an event *P* must always be preceded by an event *T* within a scope [DAC99]. The automata that represent this pattern through the SPS scopes (*global*, *before R*, *after L*, *between L and R*, and *after L until R*) are presented below.



(a) *Global* scope

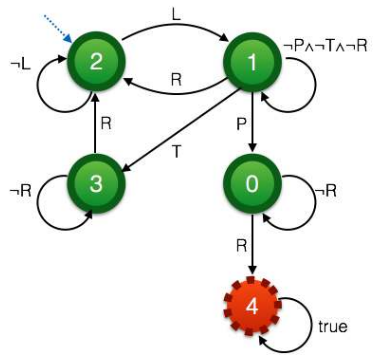


(b) *Before R* scope

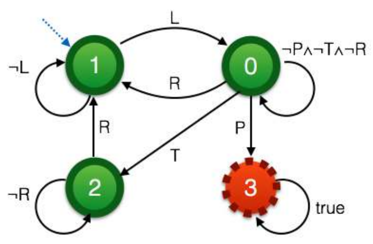


(c) *After L* scope

Figure D.5: *Precedence Pattern - DFA Collection*



(a) *Between L and R* scope

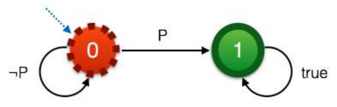


(b) *After L until R* scope

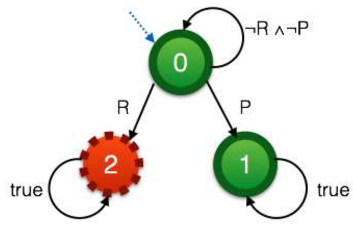
Figure D.6: *Precedence Pattern - DFA Collection (cont.)*

D.4 Existence Pattern

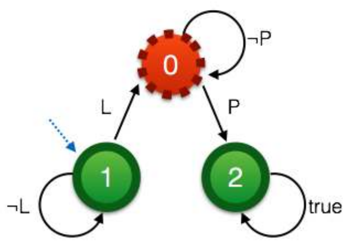
The *existence* pattern states that a given event P must occur (at least once) within a scope [DAC99]. The automata that represent this pattern through the SPS scopes (*global*, *before R*, *after L*, *between L and R*, and *after L until R*) are presented next.



(a) *Global* scope

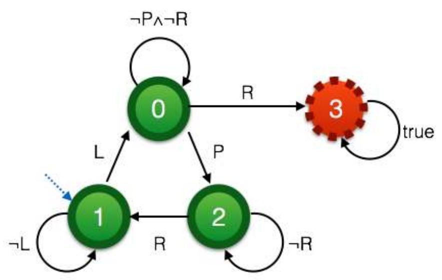


(b) *Before R* scope

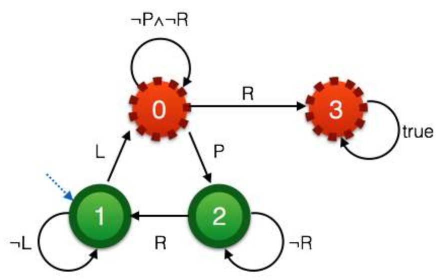


(c) *After L* scope

Figure D.7: *Existence Pattern - DFA Collection*



(a) *Between L and R* scope

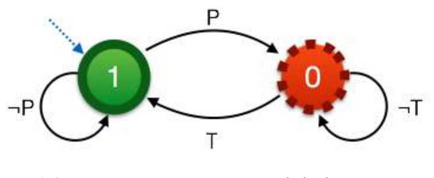


(b) *After L until R* scope

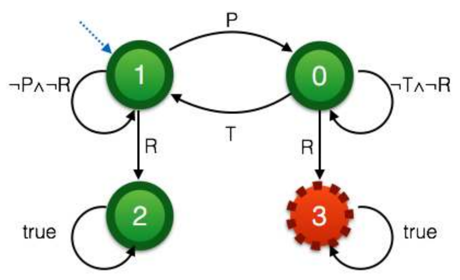
Figure D.8: *Existence Pattern - DFA Collection (cont.)*

D.5 Response Pattern

The *response* pattern says that an event *P* must always be followed by an event *T* within a scope [DAC99]. The automata that represent this pattern through the SPS scopes (*global*, *before R*, *after L*, *between L and R*, and *after L until R*) are presented below.

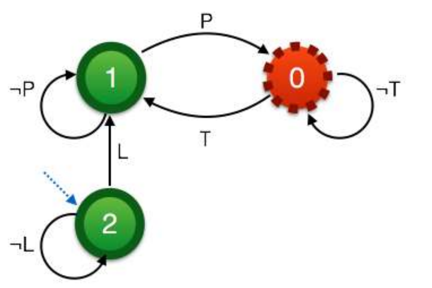


(a) *Response* pattern - *Global* scope



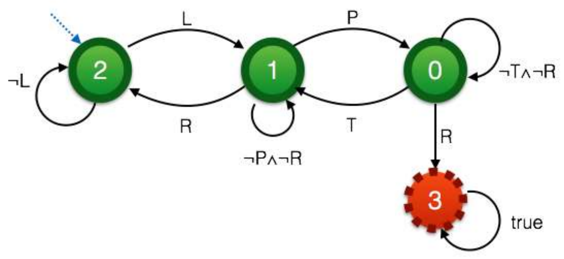
(b) *Response* pattern - *Before R* scope

Figure D.9: *Response Pattern - DFA Collection*

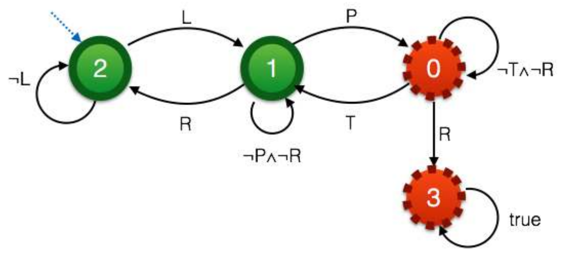


(a) *Response* pattern - *After L* scope

Figure D.10: *Response Pattern - DFA Collection (cont.)*



(a) *Response* pattern - *Between L and R* scope



(b) *Response* pattern - *After L until R* scope

Figure D.11: *Response Pattern - DFA Collection (cont.)*

⇒ The automata library that was presented in this appendix is stored and used in JPF-SPS (Chapter 6) through files. The file structures for automata representation are described in Section 5.1.3. A graphical representation of the property automaton is also provided by JPF-SPS when the property is defined and verified. It can be viewed on the verification tab of the tool.

Glossary

Finite Transition System

A finite transition system [BK08] has no terminal states (states with no successors) and is represented by a tuple (Q, Act, T, Q_0, AP, L) where:

- Q is a finite set of states;
- Act is a finite set of actions;
- $T \subseteq Q \times Act \times Q$ is a transition relation;
- $Q_0 \subseteq Q$ is a set of initial states;
- AP is a finite set of atomic propositions; and,
- $L \colon Q \to 2^{AP}$ is a labeling function.

Nondeterministic Buchi Automaton

A nondeterministic Büchi automaton (NBA) A is an automata model that recognizes languages of infinite words (or ω-regular languages). It is a tuple $A = (Q, \Sigma, \delta, Q_0, F)$ where:

- Q is a finite set of states;
- Σ is an alphabet;
- $\delta : Q \times \Sigma \to 2^Q$ is a transition function;
- $Q_0 \subseteq Q$ is a set of initial states;
- $F \subseteq Q$ is a set of *accept* (or *final*) states, called the *acceptance set*.

The accepted language of NBA A – denoted by $L_\omega(A)$ – consists of all infinite words that have a run in which some accept state is visited infinitely often [BK08].

Property

A property usually describes the behavior of an IUT, and in this case, we say that a property can be represented by a test purpose and it is satisfied if an accept state is reached, otherwise, it is violated. Although there are properties that do not describe the system behavioral specification [FFJ+10], we consider in this work only the properties that describe it and can be represented by a Test Purpose. Thus, in our work we consider system properties and test purposes as synonyms (see Section 2.2.1 for details).

Property Testability

Considering a finite test execution σ and a property p that is represented by a temporal formula φ, we can conclude that φ is *testable* if at least one of the four relations between the set of executions satisfying φ and the set of (finite or infinite) executions that could be produced by continuations of σ holds [FFJ+10, NGH93]. Naming the set of φ execution sequences as $\mathsf{Tr}(\varphi)$ and the set of the implementation under test (IUT) execution sequences as $\mathsf{Tr}(\mathsf{IUT})$, we can describe these four relations as:

R1: $\mathsf{Tr}(\mathsf{IUT}) \subseteq \mathsf{Tr}(\varphi)$;

R2: $\mathsf{Tr}(\varphi) \subseteq \mathsf{Tr}(\mathsf{IUT})$;

R3: $\mathsf{Tr}(\varphi) = \mathsf{Tr}(\mathsf{IUT})$;

R4: $\mathsf{Tr}(\varphi) \cap \mathsf{Tr}(\mathsf{IUT}) = \emptyset$.

Safety-Progress Classification

The SP classification or the Borel Hierarchy [MP90] categorizes the linear temporal properties in six classes that are organized in a hierarchical manner: *safety* (it states that some bad thing *never* happens); *guarantee* (it states that some good thing occurs *at least once*); *obligation* (it is expressed by a disjunction of safety and guarantee formulas); *recurrence* (it states that some good thing occurs infinitely often); *persistence* (it states that some good thing occurs continuously from a certain point on); and, *reactivity* (it is expressed by a disjunction of recurrence and persistence formulas).

Software Error

An incorrect internal state that is the manifestation of some fault [AO08].

Software Failure

External, incorrect behavior with respect to the requirements or other description of the expected behavior [AO08].

Software Fault

A static defect in the software [AO08].

Software Test

The process of evaluating software by observing its execution [AO08].

Specification Pattern System

Specification patterns [DAC99] are high-level, formalism independent specification abstractions defined for finite state verification. Their purpose is to assist practitioners in mapping descriptions of system behavior into their formalism of choice, improving the transition of these formal methods to practice. To define a property using the SPS, one must define its scope and then, the corresponding pattern. A brief description of scopes and patterns are presented next. For a complete description of the SPS, please refer to [DAC99].

- **Scopes.** A scope is the extent of the program execution over which the pattern must hold, and is determined by specifying a starting and an ending state or event for the pattern. They are divided in five categories: *global*; *before*; *after*; *between*; and, *after-until*.
- **Patterns.** A pattern is a specification abstraction that can be mapped to various formalisms, including LTL. The patterns are organized in a hierarchy based on their semantics. Here, we are going to describe the patterns that appear in our work: *absence* (a state or event does not occur within a scope); *existence* (a state or event must occur within a scope); *universality* (a state or event occurs throughout a scope); *precedence* (a state or event P must always be preceded by a state or event T within a scope); and, *response* (a state or event P must always be followed by a state or event T within a scope).

Test Purpose

Test purpose [dT01, JJ05] is a description of what is being tested concerning to a particular system specification requirement. It presents the desirable system behavior that we want to observe during its execution, describing the execution sequences that lead to an *accept* state and/or to a *refuse* state.

Testable Property

A property is *testable* if at least one of the four relations of the Property Testability concept holds [FFJ$^+$10, NGH93].

Validation

The process of evaluating software at the end of software development to ensure compliance with intended usage [AO08].

Verification

The process of determining whether the products of a given phase of the software development process fulfill the requirements specified during the previous phase [AO08].

Bibliography

[ABB$^+$00] Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Martin Giese, Elmar Habermalz, Reiner Hähnle, Wolfram Menzel, and Peter H. Schmitt. The KeY Approach: Integrating Object Oriented Design and Formal Verification. In *Proceedings of the European Workshop on Logics in Artificial Intelligence*, JELIA '00, pages 21–36, London, UK, 2000. Springer-Verlag.

[Ach13] Camila Achutti. Mission Critical Software Validation and Verification. Monograph, 2013.

[AO08] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, New York, NY, USA, 1 edition, 2008.

[ASST10] Islam Abdelhalim, James Sharp, Steve Schneider, and Helen Treharne. Formal Verification of Tokeneer Behaviours Modelled in fUML Using CSP. In Jin Dong and Huibiao Zhu, editors, *Formal Methods and Software Engineering*, volume 6447 of *Lecture Notes in Computer Science*, pages 371–387. Springer Berlin / Heidelberg, 2010. 10.1007/978-3-642-16901-4_25.

[Ban] Bandera. http://bandera.projects.cis.ksu.edu/. Last access: Aug. 2015.

[Bel10] Axel Belinfante. Jtorx: A tool for on-line model-driven test derivation and execution. In Javier Esparza and Rupak Majumdar, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 6015 of *Lecture Notes in Computer Science*, pages 266–270. Springer Berlin Heidelberg, 2010.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[BLS06] Andreas Bauer, Martin Leucker, and Jonathan Streit. Salt: Structured assertion language for temporal logic. In Zhiming Liu and Jifeng He, editors, *Formal Methods and Software Engineering*, volume 4260 of *Lecture Notes in Computer Science*, pages 757–775. Springer Berlin Heidelberg, 2006.

[Boe81] Barry W. Boehm. *Software Engineering Economics*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1981.

[BPN$^+$11] Daniel Balasubramanian, Gábor Pap, Harmon Nine, Gabor Karsai, Michael R. Lowry, Corina S. Pasareanu, and Thomas Pressburger. Rapid property specification and checking for model-based formalisms. In *International Symposium on Rapid System Prototyping*, pages 121–127. IEEE, 2011.

[CDJ$^+$13] Kalou Cabrera Castillos, Frédéric Dadeau, Jacques Julliand, Bilal Kanso, and Safouan Taha. A Compositional Automata-Based Semantics for Property Patterns. In EinarBroch Johnsen and Luigia Petre, editors, *Integrated Formal Methods*, volume 7940 of *Lecture Notes in Computer Science*, pages 316–330. Springer Berlin Heidelberg, 2013.

[CMP92] Edward Chang, Zohar Manna, and Amir Pnueli. The Safety-Progress Classification. Technical report, Stanford University, 1992.

[DAC99] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in Property Specifications for Finite-State Verification. In *Proceedings of the 21st international conference on Software engineering*, ICSE '99, pages 411–420, New York, NY, USA, 1999. ACM.

[DHJ+01] Matthew B. Dwyer, John Hatcliff, Roby Joehanes, Shawn Laubach, Corina S. Păsăreanu, and Hongjun Zheng. Tool-supported program abstraction for finite-state verification. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 177–187, 2001.

[DPJ08] Dino Distefano and Matthew J. Parkinson J. jStar: Towards Practical Verification for Java. *SIGPLAN Not.*, 43:213–226, October 2008.

[DRH06] Xianghua Deng, Robby, and John Hatcliff. Kiasan: A Verification and Test-Case Generation Framework for Java Based on Symbolic Execution. In *Proceedings of the Second International Symposium on Leveraging Applications of Formal Methods, Verification and Validation*, ISOLA '06, pages 137–, Washington, DC, USA, 2006. IEEE Computer Society.

[dSM06] Daniel Aguiar da Silva and Patrícia D. L. Machado. Towards Test Purpose Generation from CTL Properties for Reactive Systems. *Electronic Notes in Theoretical Computer Science*, 164(4):29–40, 2006.

[dT01] René G. de Vries and Jan Tretmans. Towards Formal Test Purposes. In J. Tretmans and H. Brinksma, editors, *Formal Approaches to Testing of Software 2001 (FATES'01)*, volume NS-01-4 of *BRICS Notes Series*, pages 61–76, Aarhus, Denkmark, August 2001.

[Ecl] Eclipse. http://www.eclipse.org/. Last access: Aug. 2015.

[EM04] Dawson Engler and Madanlal Musuvathi. Static Analysis versus Software Model Checking for Bug Finding. In Bernhard Steffen and Giorgio Levi, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 2937 of *Lecture Notes in Computer Science*, pages 191–210. Springer Berlin Heidelberg, 2004.

[End72] Herbert B. Enderton. *A mathematical introduction to logic*. Academic Press, 1972.

[FFJ+10] Yliès Falcone, Jean-Claude Fernandez, Thierry Jéron, Hervé Marchand, and Laurent Mounier. More testable properties. In *Proceedings of the 22nd IFIP WG 6.1 international conference on Testing software and systems*, ICTSS'10, pages 30–46, Berlin, Heidelberg, 2010. Springer-Verlag.

[Fow03] Martin Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[Fra92] Nissim Francez. *Program Verification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1992.

[Gab87] Dov M. Gabbay. The declarative past and imperative future: Executable temporal logic for interactive systems. In *Temporal Logic in Specification*, pages 409–448, London, UK, UK, 1987. Springer-Verlag.

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley, 2005.

[GR92] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992.

[GRS09] Angelo Gargantini, Elvinia Riccobene, and Patrizia Scandurra. Integrating Formal Methods with Model-Driven Engineering. In *Proceedings of the 2009 Fourth International Conference on Software Engineering Advances*, ICSEA '09, pages 86–92, Washington, DC, USA, 2009. IEEE Computer Society.

[Han10] Simone Hanazumi. An Integrated Environment for Verification and Test of Fault-tolerant Components Coordination (Portuguese). Master's thesis, University of São Paulo, São Paulo, Brazil, 2010.

[HD01] John Hatcliff and Matthew B. Dwyer. Using the Bandera Tool Set to Model-Check Properties of Concurrent Java Software. In *CONCUR '01: Proceedings of the 12th International Conference on Concurrency Theory*, pages 39–58, London, UK, 2001. Springer-Verlag.

[HDD+03] John Hatcliff, Xinghua Deng, Matthew B. Dwyer, Georg Jung, and Venkatesh Prasad Ranganath. Cadena: An Integrated Development, Analysis, and Verification Environment for Component-based Systems. *Software Engineering, International Conference on*, 0:160, 2003.

[HM12] Simone Hanazumi and Ana C.V. de Melo. Coordinating Exceptions of Java Systems: Implementation and Formal Verification. In *Proceedings of the 8th International Conference on the Quality of Information and Communications Technology*, QUATIC 2012, pages 108–113, Lisbon, Portugal, 2012. IEEE Computer Society.

[Hoa85] C. A. R. Hoare. *Communicating sequential processes*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1985.

[Hol97] Gerard J. Holzmann. The Model Checker SPIN. *IEEE Trans. Softw. Eng.*, 23(5):279–295, 1997.

[Jav] Java PathFinder. http://babelfish.arc.nasa.gov/trac/jpf/. Last access: Aug. 2015.

[JFL] JFLAP. http://http://www.jflap.org. Last access: Aug. 2015.

[Jin07] Ying Jin. Formal Verification of Protocol Properties of Sequential Java Programs. In *Proceedings of the 31st Annual International Computer Software and Applications Conference - Volume 01*, COMPSAC '07, pages 475–482, Washington, DC, USA, 2007. IEEE Computer Society.

[JJ05] Claude Jard and Thierry Jéron. TGV: theory, principles and algorithms: A tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems. *Int. J. Softw. Tools Technol. Transf.*, 7(4):297–315, August 2005.

[JL] JPF-LTL. http://babelfish.arc.nasa.gov/trac/jpf/wiki/summer-projects/2010-ltl-1. Last access: Aug. 2015.

[KC05a] S. Konrad and B.H.C. Cheng. Facilitating the construction of specification pattern-based properties. In *Requirements Engineering, 2005. Proceedings. 13th IEEE International Conference on*, pages 329–338, Aug 2005.

[KC05b] Sascha Konrad and Betty H. C. Cheng. Real-time specification patterns. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 372–381, New York, NY, USA, 2005. ACM.

[Lam77] L. Lamport. Proving the Correctness of Multiprocess Programs. *IEEE Trans. Softw. Eng.*, 3(2):125–143, March 1977.

[LB03] Michael Leuschel and Michael Butler. ProB: A Model Checker for B. In *FME 2003: Formal Methods*, volume 2805 of *Lecture Notes in Computer Science*, pages 855–874. Springer Berlin / Heidelberg, 2003. 10.1007/978-3-540-45236-2_46.

[LL03] Brad Long and Benjamin W. Long. Formal Specification of Java Concurrency to Assist Software Verification. In *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, IPDPS '03, pages 136.2–, Washington, DC, USA, 2003. IEEE Computer Society.

[Mat] Matlab. http://www.mathworks.com/products/simulink/index.html. Last access: Aug. 2015.

[MGR04] Oscar Mondragon, Ann Q. Gates, and Steven Roach. Prospec: Support for Elicitation and Formal Specification of Software Properties. In *Proceedings of Runtime Verification Workshop, ENTCS*, 2004.

[Mit03] Ruslan Mitkov. *The Oxford Handbook of Computational Linguistics (Oxford Handbooks in Linguistics S.)*. Oxford University Press, 2003.

[MORW04] Michael Möller, Ernst-Rüdiger Olderog, Holger Rasch, and Heike Wehrheim. Linking CSP-OZ with UML and Java: A Case Study. In *In Integrated Formal Methods, number 2999 in Lecture Notes in Computer Science*, pages 267–286. Springer, 2004.

[MORW08] Michael Möller, Ernst-Rüdiger Olderog, Holger Rasch, and Heike Wehrheim. Integrating a Formal Method Into a Software Engineering Process with UML and Java. *Form. Asp. Comput.*, 20:161–204, February 2008.

[MP90] Zohar Manna and Amir Pnueli. A hierarchy of temporal properties (invited paper, 1989). In *Proceedings of the ninth annual ACM symposium on Principles of distributed computing*, PODC '90, pages 377–410, New York, NY, USA, 1990. ACM.

[MP91] Zohar Manna and Amir Pnueli. Completing the Temporal Picture. *Theor. Comput. Sci.*, 83(1):97–130, June 1991.

[MP92] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems - Specification*. Springer-Verlag New York, Inc., New York, NY, USA, 1992.

[MP95] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems - Safety*. Springer-Verlag New York, Inc., New York, NY, USA, 1995.

[MSF06] Ana Cristina Vieira De Melo, Flavio Soares Correa Da Silva, and Marcelo Finger. *Lógica para Computação*. Thomson Learning, 2006.

[MTL10] Qaisar A. Malik, Dragos Truscan, and Johan Lilius. Using UML Models and Formal Verification in Model-Based Testing. In *Proceedings of the 2010 17th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems*, ECBS '10, pages 50–56, Washington, DC, USA, 2010. IEEE Computer Society.

[Möl02] Michael Möller. Specifying and Checking Java using CSP. Technical report, Department, University of Nijmegen, 2002.

[NGH93] Robert Nahm, Jens Grabowski, and Dieter Hogrefe. Test Case Generation for Temporal Properties. Technical report, Bern University, 1993.

[O'R08] Gerard O'Regan. *A Brief History of Computing*. Springer, 2008.

[Ora] Oracle. Java Technology. http://www.oracle.com/br/technologies/java/index.html. Last access: Aug. 2015.

[Par08] Pavel Parízek. *Formal Verification of Components in Java*. PhD thesis, Charles University, Prague, Czech Republic, 2008.

[Pat] Specification Patterns. http://patterns.projects.cis.ksu.edu/. Last access: Aug. 2015.

[PdM10] David P. Pereira and Ana C. V. de Melo. Formalization of an architectural model for exception handling coordination based on CA action concepts. *Sci. Comput. Program.*, 75:333–349, May 2010.

[PDV01] Corina S. Pasareanu, Matthew B. Dwyer, and Willem Visser. Finding Feasible Counter-examples when Model Checking Abstracted Java Programs. In *TACAS 2001: Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 284–298, London, UK, 2001. Springer-Verlag.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *SFCS '77: Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, pages 46–57, Washington, DC, USA, 1977. IEEE Computer Society.

[Pre01] Roger S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill Higher Education, 2001.

[Ran75] B. Randell. System structure for software fault tolerance. In *Proceedings of the international conference on Reliable software*, pages 437–449, New York, NY, USA, 1975. ACM.

[RDH04] Edwin Rodríguez, Matthew B. Dwyer, and John Hatcliff. Checking strong specifications using an extensible software model checking framework. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 404–420. Springer, 2004.

[Rec] Recoder. http://recoder.sourceforge.net/. Last access: Aug. 2015.

[RH05] Grigore Rocsu and Klaus Havelund. Rewriting-Based Techniques for Runtime Verification. *Automated Software Engineering*, 12(2):151–197, 2005.

[Rod06] Susan H. Rodger. *JFLAP: An Interactive Formal Languages and Automata Package*. Jones and Bartlett Publishers, Inc., USA, 2006.

[Rom01] Alexander Romanovsky. Coordinated atomic actions: how to remain ACID in the modern world. *SIGSOFT Softw. Eng. Notes*, 26(2):66–68, 2001.

[SGRE11] Salamah Salamah, Ann Q. Gates, Steve Roach, and Matthew Engskow. Towards Support for Software Model Checking: Improving the Efficiency of Formal Specifications. *Adv. Soft. Eng.*, 2011:3:1–3:13, January 2011.

[SGRM05] Salamah Salamah, Ann Gates, Steve Roach, and Oscar Mondragon. Verifying Pattern-Generated LTL Formulas: a Case Study. In *Proceedings of the 12th international conference on Model Checking Software*, SPIN'05, pages 200–220, Berlin, Heidelberg, 2005. Springer-Verlag.

[SJ11] Valdivino Alexandre Santiago Jr. *SOLIMVA: A Methodology for Generating Model-Based Test Cases from Natural Language Requirements and Detecting Incompleteness in Software Specifications*. PhD thesis, National Institute for Space Research (INPE), São Paulo, Brazil, 2011.

[Smi00] Graeme Smith. *The Object-Z Specification Language*. Springer US, 2000.

[SPI] SPIN. http://spinroot.com. Last access: Aug. 2015.

[Tre08] Jan Tretmans. Model based testing with labelled transition systems. In Robert M. Hierons, Jonathan P. Bowen, and Mark Harman, editors, *Formal Methods and Testing*, pages 1–38. Springer-Verlag, Berlin, Heidelberg, 2008.

[VHB$^+$03] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model Checking Programs. *Automated Software Engineering Journal*, 10(2), April 2003.

[WD96] Jim Woodcock and Jim Davies. *Using Z: specification, refinement, and proof*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[WFW09] Martin Weiglhofer, Gordon Fraser, and Franz Wotawa. Using coverage to automate and improve test purpose based testing. *Inf. Softw. Technol.*, 51(11):1601–1617, November 2009.

[Wor96] J. B. Wordsworth. *Software engineering with B*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.

[XRR+95] Jie Xu, B. Randell, A. Romanovsky, C.M.F. Rubira, R.J. Stroud, and Zhixue Wu. Fault Tolerance in Concurrent Object-Oriented Software through Coordinated Error Recovery. *FTCS '95: Proceedings of the Twenty-Fifth International Symposium on Fault-Tolerant Computing*, 0:0499, 1995.

[YP07] Letu Yang and Michael Poppleton. Automatic Translation from Combined B and CSP specification to Java Programs. In Jacques Juilland and Olga Kouchnarenko, editors, *7th International B Conference*, volume 4355 of *Lecture Notes in Computer Science*, pages 64–78. Springer-Verlag Lecture Notes in Computer Science, January 2007.

[ZBdC+11] Edgardo Zoppi, Víctor Braberman, Guido de Caso, Diego Garbervetsky, and Sebastián Uchitel. Contractor.net: Inferring typestate properties to enrich code contracts. In *Proceedings of the 1st Workshop on Developing Tools As Plug-ins*, TOPI '11, pages 44–47, New York, NY, USA, 2011. ACM.

COLOPHON

This document was typeset using adapted versions of the following LaTeX templates:

- “IME-USP: Dissertation/Thesis LaTeX Example” developed by Jesús P. Mena-Chalco. This template was downloaded from: http://www.vision.ime.usp.br/~jmena/stuff/tese-exemplo/

- “The Legrand Orange Book” - Version 1.3 (21/8/13) developed by Mathias Legrand under the License: CC BY-NC-SA 3.0[1]. This template was downloaded from: http://www.LaTeXTemplates.com. In its adapted version, an image that is freely available at http://lzamgs.com/abstract-blue-background-hd-pictures-4-hd-wallpaper.html was edited and used to compose the background of parts and chapter titles of this document.

[1]http://creativecommons.org/licenses/by-nc-sa/3.0/