Model checking techniques for runtime testing and QoS analysis

Alberto Salmerón Moreno

Supervised by Pedro Merino Gómez

Department of Computer Science

University of Málaga

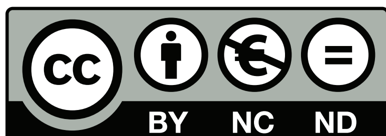
May 2014

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

Publicaciones y
Divulgación Científica

AUTOR: Alberto Salmerón Moreno

EDITA: Publicaciones y Divulgación Científica. Universidad de Málaga



Esta obra está sujeta a una licencia Creative Commons:
Reconocimiento - No comercial - SinObraDerivada (cc-by-nc-nd):
Http://creativecommons.org/licences/by-nc-nd/3.0/es
Cualquier parte de esta obra se puede reproducir sin autorización
pero con el reconocimiento y atribución de los autores.
No se puede hacer uso comercial de la obra y no se puede alterar, transformar o hacer
obras derivadas.

Esta Tesis Doctoral está depositada en el Repositorio Institucional de la Universidad de Málaga (RIUMA): riuma.uma.es

Publicaciones y
Divulgación Científica
ma
UNIVERSIDAD
DE MÁLAGA

El Dr. Don Pedro Merino Gómez, Titular de Universidad del Área de Telemática de la E.T.S. de Ingeniería de Telecomunicación de la Universidad de Málaga,

Certifica que Don Alberto Salmerón Moreno, Ingeniero en Informática, ha realizado en el Departamento de Lenguajes y Ciencias de la Computación de la Universidad de Málaga, bajo mi dirección, el trabajo de investigación correspondiente a su Tesis Doctoral titulada:

Model checking techniques
for runtime testing and QoS analysis

Revisado el presente trabajo, estimo que puede ser presentado al tribunal que ha de juzgarlo, y autorizamos la presentación de esta Tesis Doctoral en la Universidad de Málaga.

Málaga, Mayo de 2014

Fdo.: Pedro Merino Gómez
Titular de Universidad
Área de Telemática

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

Acknowledgements

This thesis has been partially funded by Andalusian projects P07-TIC-03131 and P11-TIC-07659, Spanish projects TIN2012-35669, TIN 2008-05932, 8.06/5.47.3154-1 and IPT-2011-1034-370000, and ERDF from the European Commission.

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

Publicaciones y
Divulgación Científica
ma
UNIVERSIDAD
DE MÁLAGA

Agradecimientos

Quiero comenzar agradeciendo a Pedro, mi director, sin el cual no existiría este volumen. Él me dio una oportunidad al acogerme en el 3.3.3, y siguió confiando en mí para la realización de este trabajo. Gracias a él he conocido los entresijos del mundo de la investigación, y he podido participar en proyectos interesantes y también desafiantes.

También quiero agradecer a María del Mar su ayuda con los aspectos más formales de este trabajo, en los que siempre que ha costado entrar, y a Almudena y Damián por su colaboración en algunos de los trabajos aquí presentados.

A todos los compañeros presentes y pasados del 3.3.3 (ahora A.1.3), porque trabajar en tan buena compañía no es trabajar. Con ellos he pasado tantos buenos momentos que darían para escribir otra tesis.

A todos los amigos con los que he vivido tantas experiencias a lo largo de estos años, y que le dan perspectiva a la vida.

A mi familia por su apoyo, aunque no lo demuestre tanto como les gustaría.

Esta tesis no habría sido la misma sin las personas con las que he compartido estos últimos años, dentro y fuera del trabajo.

A todos vosotros, gracias.

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

Contents

Contents i

List of Figures v

List of Tables vii

List of Listings ix

1 Introduction 1
- 1.1 Overview . . . 2
- 1.2 Contributions . . . 4
- 1.3 Organization . . . 4

I Preliminaries 7

2 State of the art 9
- 2.1 Model checking . . . 10
  - 2.1.1 System modeling . . . 11
  - 2.1.2 Property specification languages . . . 11
  - 2.1.3 Automata-based LTL model checking . . . 13
  - 2.1.4 Abstraction . . . 16
  - 2.1.5 The Spin model checker . . . 19
- 2.2 Trace-based analysis . . . 23
  - 2.2.1 Analysis of network simulations . . . 24
  - 2.2.2 Analysis of Java programs . . . 27

II The OptySim trace analysis framework 31

3 Analysis framework 33
- 3.1 Framework overview . . . 33

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

3.1.1 Setup . . . 34
3.1.2 Code generation . . . 36
3.1.3 Analysis . . . 37
3.1.4 Results . . . 38
3.2 Protocol . . . 39
3.2.1 Negotiation . . . 40
3.2.2 Setup . . . 42
3.2.3 Execution . . . 43
3.2.4 Example . . . 44
3.3 Objectives . . . 44
3.3.1 State assertions . . . 47
3.3.2 LTL formulas and never claim automata . . . 47
3.3.3 Parameter configuration validation . . . 48
3.3.4 Inferring results from previous analyses . . . 49
3.4 Implementation . . . 50
3.4.1 Promela template . . . 50
3.4.2 Data structures . . . 51
3.4.3 Exploration and analysis . . . 55
3.4.4 Protocol implementation . . . 57
3.4.5 Template instantiation . . . 60
3.4.6 System integration . . . 61
3.5 Summary . . . 61

4 Abstraction of execution traces 63
4.1 State space . . . 64
4.1.1 Parameter space . . . 65
4.1.2 External factors . . . 66
4.2 Trace projection . . . 68
4.2.1 Counter projection . . . 70
4.2.2 Hash projection . . . 71
4.2.3 Folded projections . . . 72
4.3 Preservation of results . . . 74
4.4 Operational semantics of executions exploration . . . 76

III Applications 85

5 Analysis of ns-2 network simulations 87
5.1 The ns-2 network simulator . . . 87
5.1.1 Architecture . . . 88
5.1.2 State space and traces . . . 90

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

5.2 Extraction of ns-2 execution traces 91
5.2.1 Overview of integration with ns-2 92
5.2.2 Generic C++ library 93
5.2.3 ns-2 integration library 96
5.3 Case study: video streaming over TCP on mobile environments 98
5.3.1 Network scenario 98
5.3.2 Reliability analysis 99
5.3.3 Performance analysis 103
5.4 Case study: E-model extension 109
5.4.1 E-model for VoIP QoE evaluation 109
5.4.2 On-the-fly E-model computation 114
5.4.3 Implementation in ns-2 115
5.4.4 Validation analysis 118

6 Inverse modeling of network KPIs 121
6.1 Key performance indicators 121
6.2 Inverse modeling methodology 122
6.2.1 SymPA 124
6.3 Case study: jitter model 124
6.3.1 Field test scenario 125
6.3.2 Replaying jitter traces in simulations 126
6.3.3 Inverse modeling 126
6.4 Related work 129

7 Analysis of Java execution traces 131
7.1 The Java programming language 131
7.1.1 Java Platform Debugger Architecture 132
7.2 Extraction of Java execution traces 133
7.2.1 Eclipse plug-in 133
7.2.2 Runtime monitor 136
7.3 Case studies 137
7.3.1 Applications and requirements 137
7.3.2 Counter projection 142
7.3.3 Hash projection 142
7.4 Related work 144
7.4.1 Comparison with JPF 144

IV Final remarks 145

8 Conclusions and Future Work 147

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

8.1 Conclusions 147
8.2 Future work 149

V Appendices 151

Appendix A: Configuration files 153
A.1 XML Schema 153
A.2 Legacy configuration 158
A.2.1 ns-2 legacy configuration 158
A.2.2 TJT legacy configuration 159

Appendix B: Protocol definitions 161
B.1 Promela model 161
B.2 Protocol buffer message definitions 169

Appendix C: Improved E-model algorithm 173

Appendix D: Resumen en español 177
D.1 El marco de trabajo de análisis OptySim 178
D.1.1 Visión general del análisis 178
D.1.2 Objetivos 180
D.2 Proyección de trazas de ejecución 181
D.3 Integración y casos de estudio: ns-2 182
D.3.1 Caso de estudio: vídeo sobre TCP 182
D.3.2 Caso de estudio: validación de E-model 183
D.3.3 Caso de estudio: modelado inverso de KPIs 183
D.4 Integración y casos de estudio: Java 185
D.5 Conclusiones 186

References 189

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

List of Figures

1.1 Overview of the OptySim analysis framework 3

2.1 Example LTL formulas 14
2.2 Transformation of a Kripke structure into a Büchi automaton 15
2.3 Nested depth first search (NDFS) algorithm 17
2.4 Overview of the Spin tool workflow for checking LTL properties 20
2.5 Data structures used by Spin 24
2.6 Overview of Verisim workflow 25

3.1 Overview of the OptySim analysis framework 34
3.2 Main loop of Spin executing the Promela template 38
3.3 Communication protocol overview: main activities 41
3.4 Communication protocol overview: execution subactivities 43
3.5 Example scenario of the communication protocol 45
3.6 Overview of the Promela template contents 52
3.7 State stack 54
3.8 Variables table 54
3.9 Generating bindings from protocol buffer message definitions 59
3.10 Overview of the Promela instantiation process 60

4.1 Trace projection 69
4.2 Trace projection with state counting 71
4.3 Trace projection with state hashing 72
4.4 Example of the folded and limited folded hash projections of a trace . . 74
4.5 Selected verification-level rules from [64]. 78
4.6 Rules for depth-first exploration of trace-based state spaces (I). 79
4.7 Rules for depth-first exploration of trace-based state spaces (II). 80
4.8 Trace exploration example 82

5.1 Event scheduling and handling example in ns-2 90
5.2 ns-2 trace: new state after each event vs. after a relevant event 91
5.3 Components of ns-2 integration 92

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

5.4 Main classes of liboptysim-cpp 94
5.5 Partial sequence diagram of protocol implementation 96
5.6 Main classes of liboptysim-cpp-ns2 97
5.7 Overview of video download over TCP 99
5.8 State diagram of the video client 101
5.9 Sample trace output for reliability objective violation 102
5.10 Influence of parameters in the experiments 108
5.11 Influence of the precision of variables 109
5.12 Comparison of adaptive vs. non-adaptive of video download throughput 110
5.13 4-state Markov model of bursty packet loss 112
5.14 Simplified state machine of the improved algorithm for burst modeling . 115
5.15 Class diagram of part of the E-model module 116
5.16 Class diagram of the new error models 118

6.1 Overview of methodology for inverse modeling of KPIs. 123
6.2 Jitter and bandwidth captured during one of the HSDPA field tests. . . . 125
6.3 Jitter from field test data vs simulation with replay 127
6.4 Simulation scenario for inverse modeling of jitter. 128
6.5 Comparison of different jitter models with a real jitter trace. 130

7.1 Overview of the Java Platform Debugger Architecture (JPDA) 132
7.2 Overview of the analysis framework 134
7.3 Screenshot of the TJT Eclipse plug-in 134

D.1 El marco de trabajo de análisis OptySim 179
D.2 Reconstrucción de trazas de ejecución en Spin 180
D.3 Metodologia para el modelado inverso de KPIs 184
D.4 Captura de pantalla del *plug-in* para Eclipse 185

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

List of Tables

3.1 Summary of supported objectives. 46

5.1 Scenario parameters and their possible values for the case study 104
5.2 Results for the experiments using regular TCP 105
5.3 Parameter selection for some link configurations 107
5.4 Validation results for our E-model extension 120

6.1 Probability distributions for modeling jitter. 129

7.1 LTL formulas used as testing objectives 138
7.2 Test results using folded counter and hash projections 143
7.3 Results of tests using TJT and JPF-LTL 144

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

List of Listings

2.1 Never claim automata for the LTL formula $\Box\Diamond p$ 19
2.2 Example Promela specification 21
2.3 Example Promela specification with embedded C code 23
2.4 Example TestNG unit test for a Fibonacci class 29
3.1 Promela template: global state 53
3.2 Promela template: struct state for state stack 53
3.3 Promela template: initialization and exploration 56
3.4 Promela template: configuration generation 57
3.5 Promela template: trace reconstruction and analysis 58
5.1 ns-2 basic scheduling algorithm implementation 89
5.2 Part of video client implementation in ns-2 100
5.3 Clark's algorithm for counting state transitions of bursty packet loss model 113
7.1 FTP server: main loop 139
7.2 FTP server: CWD command 140
7.3 Elevator: elevator waiting for clients 141
A.1 XML Schema definition of configuration files 153
B.1 Promela model of communication protocol 161
B.2 Protocol buffers definitions for communication protocol 169
C.1 Improved state transition counting algorithm; packet reception event . . 174
C.2 Improved state transition counting algorithm; packet loss event 175

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

LIST OF LISTINGS

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

Chapter 1

Introduction

Hardware and software systems are increasingly present and vital in our lives. From traditional desktop computers to tiny mobile devices and huge mainframes, and from simple user applications to controller systems and network stacks.

Developing reliable and efficient systems is a hard task which is fundamental for many critical areas. Thus, systems have to be analyzed to check that they comply with what is required of them. For some systems it is enough to check that no unexpected behaviors are produced during normal operation. For others, it may require optimizing the available system parameters to reach a satisfactory performance.

Many analysis techniques and tools have been proposed to deal with this problem. Different types of systems and stages of the development process have been traditionally the subject to different techniques. For instance, assessing the performance of a networked system may be done analytically at first, then using network simulations during development, and finally through measurement tools in the final deployment stage.

Model checking [45] is one such analysis technique, which has been successfully been applied to both software and hardware systems. Model checkers are tools that analyze the full state space of a system, i.e. all its possible behaviors, to check if a given property is true. If an error is found, the model checker provides a counter-example, i.e. a trace that shows a path from the start of the system to the place the error was found on, which assists the developers in finding and fixing the error.

Although model checking has been traditionally associated with the analysis of models of systems, it has also been applied to the analysis of real systems, such as C o Java programs [31][48][74]. One of the most well known issues with model checking is the state space explosion problem: the size of the state space to be explored grows quickly with the complexity of the system, and may end up being too large to be covered in practice. Several techniques have been proposed to address this problem, such as abstraction or partial order reduction.

One of the most extended techniques for analyzing software systems during development is testing [28]. Testing techniques can be broadly classified into white-boxing and

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

black-boxing techniques. The former assume knowledge of the implementation of the system and has access to its components, while the latter does not and is only aware of the "output" of the system. Testing can also be done at several levels. For instance, in unit testing [30][21] small parts of a system, such as a class or even just a function, are tested in isolation. Higher levels test the integration of some of the components, or the system as a whole.

These techniques have their strengths and weaknesses. While model checking is a powerful technique that allows the definition of complex temporal requirements, the state space explosion problem is a significant hurdle for its adoption in complex systems. In many cases, instead of analyzing the complete state space, it may be useful to analyze a subset, such as a studying a significant number of system executions.

Although their underlying concepts are the same, many tools are tailored specifically for a type of system, or a type of analysis within a system. For instance, for analyzing the correctness of a network protocol, a developer may use model checking and testing tools, while the performance evaluation and optimization may be performed using a network simulator. This approach has several drawbacks. On the one hand, the user must learn to use several tools, with potentially different ways for expressing requirements. On the other hand, several artifacts or models may be required for analyzing a single system. These must be kept in sync, and even then there may be a mismatch between the level of detail accepted by each which may devalue the usefulness of the analyses.

1.1 Overview

In this thesis, we propose an approach to use model checking to analyze external systems whose behavior can be observed as execution traces, i.e. ordered sequences of states, called OptySim. External systems are accessed in a uniform way, enabling the use of a single tool for many types of systems. We show that this integration is also useful for many purposes, from testing to performance optimization, through a series of case studies in different areas.

Instead of analyzing the whole state space of a system, which may be impractical or unavailable, we only deal with the subset given by a set of execution traces. A number of executions are carried out, possibly by altering the values of a set of system parameters, and a trace is generated for each of them. The content of these traces depend on the system itself, and may be adapted depending on the needs of the analysis. We present two projections that transform a full trace, i.e. one that contains the full state information for each state, into an abstract trace containing fewer states and/or less information per state.

The analysis is guided by one or more objectives provided by the user, which convey the meaning of the analysis that the user wants to carry out. Objectives may we written in different forms, such as simple state assertions or linear temporal logic (LTL) formulas.

Publicaciones y Divulgación Científica

UNIVERSIDAD DE MÁLAGA

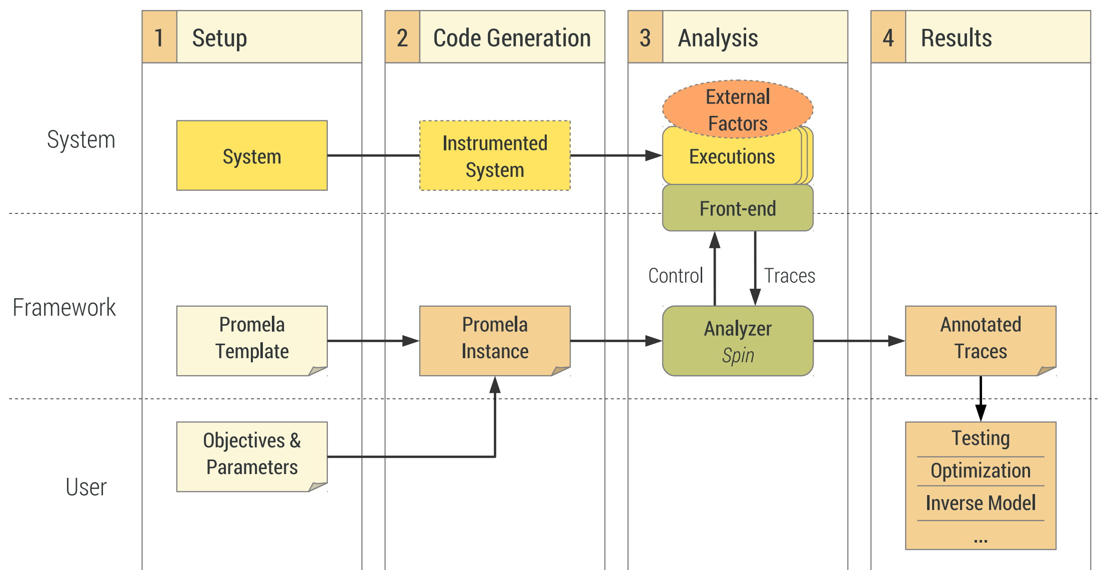


Figure 1.1: Overview of the OptySim analysis framework

An objective may indicate a desired property of the system, e.g. a performance objective, or a undesired property, e.g. an error condition.

In many types of analysis, it is enough to analyze a trace up until an objective has been met, instead of reporting every property violation within a trace, as it is custom in model checking. In our approach, execution traces are analyzed on-the-fly, i.e. while the system is still running. This on-the-fly analysis allows us to stop executions early when the analysis outcome for an execution trace is known, and can lead to significant resource savings.

Figure 1.1 shows an overview of our analysis framework. Vertical boxes separate the different steps of the analysis process, while horizontal lines delineate the responsibilities of the user, the system being analyzed and the framework itself. In the first step, the user provides the system, plus the objectives to check. If the system is parameterized, the user can provide value ranges for them so that the system can be analyzed using different parameter configurations. In the next step the analyzer is generated using this information from the user. The third step contains the bulk of the analysis. Using a custom protocol, the analyzer communicates with the system (or a front-end developed for a particular type of system) to obtain execution traces and control other aspects of the execution. Finally, a set of annotated traces is returned, declaring which objective was met in each trace, if any.

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

1.2 Contributions

The main contributions of this thesis, and the publications where they were presented, can be summarized in the following points:

- A framework for analyzing execution traces using model checking [93][23]. This framework is based on the Spin model checker and a Promela template which is customized before each analysis. Among other capabilities, it includes support for generating parameter configurations and trace backtracking.
- A protocol for interacting with an external system and extract execution traces on-the-fly. This protocol allows the separation of the analyzer and the system under analysis, and can be reused for other systems through an appropriate front-end.
- An extension for controlling the exploration of state spaces composed of independent traces. The first available result is returned for each trace and the analysis proceeds with the next one, reducing the state space to be analyzed.
- Two abstraction methods for reducing the information required in execution traces [23][22]. These methods, called counter and hash projections, minimize the amount of information per state. The hash projection also allows cycle detection. The use of folding further reduces the number of states in a trace.
- Integration with the ns-2 network simulator, applied to correctness and performance analysis [93][92], validation [105], and KPI modeling [55].
- An extension of Clark's E-model algorithm for estimating VoIP call quality on-the-fly [105].
- Integration with Java and Eclipse for testing Java programs [23][22]. Using the Java debugging interface (JDI), execution traces can be extracted without modifying the original Java program. The use of the hash projection allows using LTL formulas that require cycle detection in tests.

1.3 Organization

The rest of thesis is organized into five parts.

Part I covers relevant state of the art for our work, from model checking, which is the backbone of our approach, to other analysis tools that have similarities to ours.

Part II describes our OptySim framework for analyzing traces from external systems. Chapter 3 starts with an overview of our framework, and then describes in more detail two important pieces: the communication protocol between our analyzer and the external

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

system, and the objectives that can be used to guide the analysis. The chapter concludes with a description of the implementation of the analyzer. Chapter 4 formalizes some of the concepts introduced in the previous chapters, such as the state space to be analyzed, and describes the trace projections supported by our framework.

Part III contains several applications of our framework to different fields and for different purposes, divided into three chapters. Chapter 5 describes our integration with the ns-2 network simulator and two case studies: video streaming over TCP and VoIP QoE evaluation. Chapter 6 addresses the problem of modeling factors such as *jitter* for performing realistic simulations. Finally, Chapter 7 describes our work towards analyzing Java program execution traces for testing. These chapters also provide an overview of the work required to integrate ns-2 and Java with our framework.

Part IV concludes the main body of this thesis, a points to future work.

Finally, Part V contains the appendices. Appendix A describes the format of the configuration files for the analysis. Appendix B contains a Promela model of the communication protocol and the complete definition of its messages. Appendix C shows the complete E-model algorithm, referenced in Chapter 5. Appendix D contains a summary of this thesis in Spanish.

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

Part I

Preliminaries

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

Publicaciones y
Divulgación Científica
UNIVERSIDAD
DE MÁLAGA

Chapter 2

State of the art

The development of a system is a complex task that must be supported by appropriate processes and tools to ensure a successful result. Systems usually have to conform to a certain design, or operate within given performance objectives, requiring proper analysis to correct errors or adjust exposed parameters.

In the context of software systems, there are many analysis tools that can be applied at different stages of development and with different purposes. Some tools are oriented to detecting errors early on during the design phase, while others are concerned with the implementation of these designs. Another class of tools is oriented towards analyzing and optimizing the performance of a system during deployment, studying the behavior of a piece of software in its real setting. Different languages and situations also lend themselves to different kinds of support from analysis tools. Having access to the source of a software component can yield more insightful analyses, but in some circumstances the only viable option is treating them as "black-boxes" with certain observable behavior.

Model checking is one such analysis technique. Model checking is a rigorous approach for verifying system models against their specifications. Its foundations were developed by Edmund Clark, Allen Emerson and Joseph Sifakis [41][104], which earned them the ACM Turing Award in 2007 [42]. Model checking is widely used in the hardware and software industries, in areas where ensuring the correctness of systems with respect to their specifications is critical.

In this chapter we will review the foundations of model checking and algorithms for checking linear temporal logic (LTL) formulas. We will also introduce the Spin model checker, a popular LTL model checker which we use as part of our analysis framework. In the second half of this chapter we will cover analysis tools for software systems, paying special attention to those with similar applications to the ones considered in Part III.

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

2.1 Model checking

Model checking is an automatic technique for verifying finite state concurrent systems [45]. Given a specification for a system, the model checker finds out whether the system meets that specification. If a violation if found, the model checker provides a counterexample, i.e. an execution trace which leads to the point where the violation was found, to help uncover the source of the error. Model checkers work with formal models of the systems to be analyzed, which in many cases can be automatically extracted. These models should capture enough detail to be able to check the desired specification, while abstracting away unimportant details.

Model checking has been applied to several areas, ranging from circuit design to communication protocols. The Spin model checker [80], for instance, has been used to check software on board a number of space missions, such as a multi-threaded plan execution module [72] or a resource arbiter in the Mars Exploration Rovers [78]. It has also been employed in the verification of the control algorithm of a flood control barrier in the Netherlands [83], and in controller generation for dam management [61]. The Uppaal model checker [89] has been used to debug an audio/video communication protocol [73], for instance. The PRISM model checker [88] has been used in a wide array of fields such as security [99] and biology [76].

In addition to simple state assertions, it is common to use *temporal logic* with a model checker tool to describe the expected behavior of a system. Temporal logic allows the user to reason about the evolution of the system over time, without dealing with time explicitly. For instance, the linear-time temporal logic (LTL) formula $\Box\Diamond p$ states that the boolean proposition p should be true an infinite number of times along a single execution path.

Simulation and testing are two popular techniques for analyzing hardware and software systems. In the former, a model of the system is studied in a simulated and controllable environment, while testing works with the real system. In both cases, the general procedure involves providing a number of inputs, and comparing the outputs with the expected results. In contrast, model checking explores the model exhaustively to check if a given specification is true or not.

Given a finite state system and enough resources, a model checker always finishes with a definite *yes* or *no* answer. However, a complex system may have a state space which is too large to be explored and analyzed for practical purposes. This is known as the *state space explosion* problem. There are several proposals that address this problem. *Partial order reduction* is a technique that can be applied to concurrent models, where exploring different orderings of concurrent transitions lead to a high number of states. Some of these orderings concern independent actions, which produce the same outcome no matter the particular ordering. Thus, only one such ordering has to be explored. Later in this section we will describe *abstraction* techniques, which may also help with the state space explosion problem.

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

2.1.1 System modeling

In order to analyze a system, a model checker needs a formal model of the system. Different model checker tools accept different formalism as input, with different abstraction levels and features. For instance, Promela is the language that describes the models accepted by the Spin model checker [80], and provides features such as processes, communication channels and indeterminism. The CMU SMV [40] and NuSMV [38] symbolic model checkers both use similar modeling languages, with modules that can be composed synchronously or asynchronously, and indeterminism. In the language supported by the PRISM model checker [88], state transitions have an associated transition probability, which enables richer forms of quantitative analysis.

Models can be constructed by hand, either from the system design or from its implementation, or extracted automatically [81]. Promela supports embedding C code, which eases the extraction of models from C source code. BLAST [31] can extract and iteratively refine a model from C source code. Bandera [48] transforms Java programs into Promela models with embedded C code, to be analyzed with Spin. On the other hand, JPF [74] can analyze unmodified Java programs by running them in a custom virtual machine.

These models can also be described with a formalism called Kripke structures [45]. A Kripke structure consists of states, transitions between states, and a labeling function for states, where labels indicate the subset of properties which are true in each state.

Definition 2.1 (Kripke structure) *Let AP be a set of atomic propositions. A Kripke structure is a tuple $M = \langle S, S_0, R, L \rangle$ where*

- *S is a finite set of states,*
- *$S_0 \subseteq S$ is the set of initial states,*
- *$R \subseteq S \times S$ is a transition relation that must be total, i.e. for every state $s \in S$ there is a state s' such that $R(s, s')$,*
- *$L : S \to 2^{AP}$ is a function that labels each state with the set of atomic propositions that are true in that state.*

2.1.2 Property specification languages

To describe the desired properties of a system, several formal languages have been proposed. One of the most powerful choices is some form of temporal logic. These logics allow reasoning about the evolution of a system over time, without mentioning time explicitly. For instance, a formula might declare that a given system state is eventually reached, or that it is repeated infinitely often in an execution. These properties are built

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

using temporal operators, such as *always* ($\Box$) or *eventually* ($\Diamond$), combining them with regular boolean operators such as *and* ($\wedge$) or *not* ($\neg$).

Temporal logics can be classified into two broad groups according to how they handle branching in the computational model being analyzed: *branching-time* logic and *linear-time* logic. Computational tree logic (CTL) [43] and linear-time temporal logic (LTL) [102] are two widely used logics of these groups, respectively, while CTL* is a superset of both logics, combining both branching-time and linear-time semantics.

CTL*

CTL* formulas describe properties of *computation trees*. Given a Kripke structure and an initial state, a tree is constructed by unwinding the structure into an infinite tree, starting with the given state at the root.

CTL* formulas are composed of *path quantifiers* and *temporal operators*. There are two path quantifiers, $\forall$ ("for all computational paths") and $\exists$ ("for some computational path"). On the other hand, there are five temporal operators, briefly explained below:

- **next ($\bigcirc p$).** Property p holds in the next state of the path.
- **always ($\Box p$).** Property p must hold at every state on the path.
- **eventually ($\Diamond p$).** Property p must hold at some state on the path.
- **until ($p\,\mathbf{U}\,q$).** It holds if property q holds at some state on the path, and every preceding state holds p.
- **release ($p\,\mathbf{R}\,q$).** It holds if property q holds at every state on the path up to and including the state in which p holds. However, it is not required for p to hold eventually.

There are two types of formulas in CTL*: *state formulas* and *path formulas*. The truth value of state formulas is determined on each state, while path formulas are evaluated along a specific path. Given a set of atomic propositions AP, the syntax rules for state formulas are as follows:

- If $p \in AP$, then p is a state formula.
- If f and g are state formulas, then $\neg f$, $f \wedge g$ and $f \vee g$ are state formulas.
- If f is a path formula, then $\exists f$ and $\forall f$ are state formulas.

On the other hand, the syntax rules for path formulas are:

- If f is a state formula, then f is also a path formula.

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

- If f and g are path formulas, then $\neg f$, $f \wedge g$, $f \vee g$, $X f$ $\Diamond f$, $\Box f$, $f\ U\ g$ and $f\ R\ g$ are path formulas.

CTL is a subset of CTL* in which temporal operators must be immediately preceded by a path quantifier.

Linear-time Temporal Logic

Linear-time temporal logic or linear temporal logic (LTL) [102] can be defined as a subset of CTL* where formulas are always of the form $\forall f$, where f is a path formula that only has atomic propositions as state subformulas. More precisely, given a set of atomic propositions AP, the syntax rules for LTL path formulas are:

- If $p \in AP$, then p is also a path formula.
- If f and g are path formulas, then $\neg f$, $f \wedge g$, $f \vee g$, $X f$ $\Diamond f$, $\Box f$, $f\ U\ g$ and $f\ R\ g$ are path formulas.

We assume that given a state σ and an atomic proposition $p \in AP$, $\sigma \models p$ represents the result of evaluating p on σ, that is, $\sigma \models p$ holds iff σ satisfies p. In what follows, given a (possibly infinite) trace $t = \sigma_0 \to \sigma_1 \cdots$, we denote with $t_i = \sigma_i \to \cdots$ the suffix of t starting at state σ_i. Consider $p \in AP$, and f and g two LTL formulas. We inductively define $\models$ over traces and LTL formulas as follows.

$$
\begin{array}{lcl}
t_i \models p & \Leftrightarrow & \sigma_i \models p \\
t_i \models \neg p & \Leftrightarrow & \sigma_i \not\models p \\
t_i \models p \vee q & \Leftrightarrow & (t_i \models p) \vee (t_i \models q) \\
t_i \models \bigcirc f & \Leftrightarrow & t_{i+1} \models f \\
t_i \models \Box f & \Leftrightarrow & (\sigma_i \models f) \wedge (t_{i+1} \models \Box f) \\
t_i \models \Diamond f & \Leftrightarrow & \exists j \geq i.(t_j \models f) \\
t_i \models f\ U\ g & \Leftrightarrow & \exists j \geq i.((t_j \models g) \wedge (\forall i \leq k < j.[t_k \models f]))
\end{array}
$$

Note that the operator *implies* “$\to$" is usually omitted in these rules and transformed into a combination of negation and disjunction. Figure 2.1 contains some example LTL formulas using these operators, with computation traces which satisfy them.

2.1.3 Automata-based LTL model checking

The choice of property specification language dictates the model checking algorithms that can be used. In particular, there are several model checking algorithms for properties written with LTL [116][44][65]. In this section we will describe an efficient automata-based approach, where the state space to be explored can be constructed and analyzed on-the-fly.

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

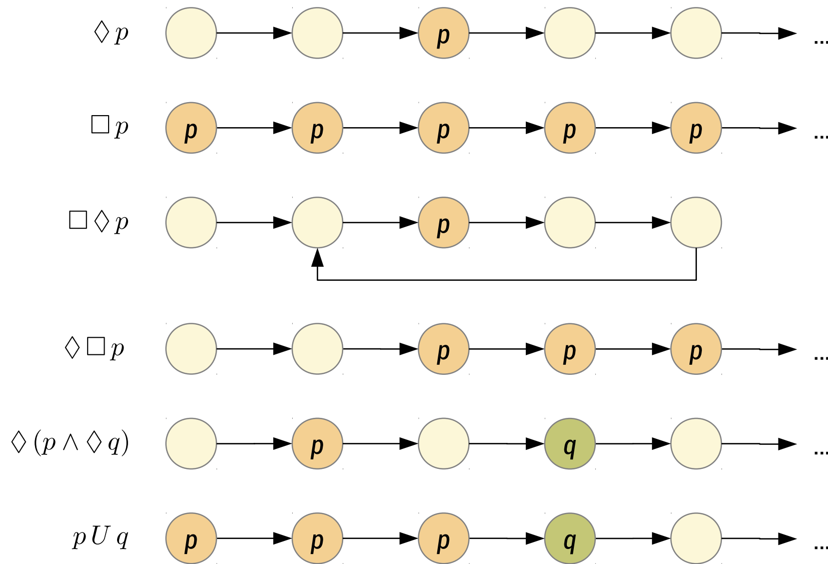


Figure 2.1: Example LTL formulas

The problem of model checking of temporal formulas can be reduced to one of reachability: given two automatas, one representing the system and another the negation of temporal formula to verify, check if the product of both is non-empty [49]. If the product is found to be non-empty, then the model checking algorithm returns a counterexample. The automatas used in this solution are Büchi automatas.

Definition 2.2 (Büchi automaton) *A Büchi automaton is a tuple $\mathcal{A} = \langle \Sigma, S, T, S_0, F \rangle$, where:*

- *Σ is a finite alphabet,*
- *S is a finite set of states,*
- *$T \subseteq S \times \Sigma \times S$ is a transition relation,*
- *$S_0 \subseteq S$ is a set of initial states,*
- *$F \subseteq S$ is a set of accepting states.*

A Büchi automaton is a finite automaton over infinite words. The language accepted by a Büchi automaton consists of all the infinite words where an accepting state is visited infinitely often.

Kripke structures, used to describe systems, can be translated into Büchi automata. A Kripke structure $M = \langle S, S_0, R, L \rangle$ where $L : S \rightarrow 2^{AP}$, can be transformed into a Büchi automaton $\mathcal{A} = \langle \Sigma, S \cup \{\iota\}, T, \{\iota\}, S \cup \{\iota\} \rangle$, where $\Sigma = 2^{AP}$. The transition relation T is

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

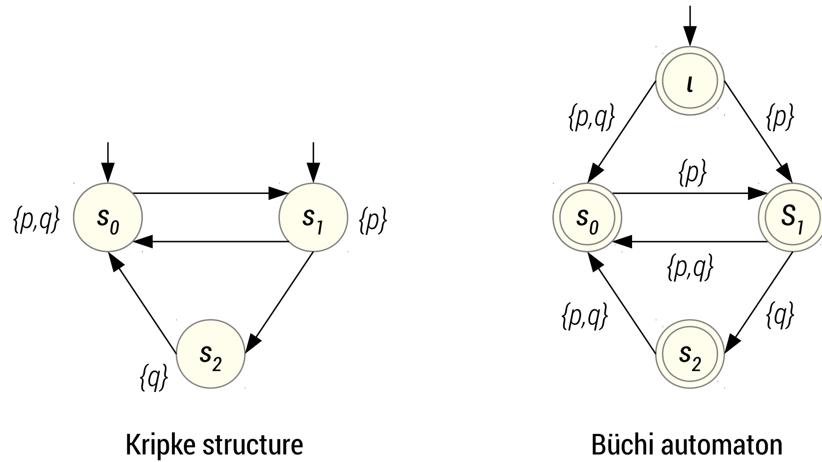


Figure 2.2: Transformation of a Kripke structure into a Büchi automaton

built as follows. Given two states s, $s' \in S$, $(s,\alpha,s') \in T$ iff $(s,s') \in R$ and $\alpha = L(s')$. In addition, $(\iota,\alpha,s) \in T$ iff $s \in S_0$ and $\alpha = L(s)$. The language accepted by this automaton, $\mathcal{L}(\mathcal{S})$, represents the set of possible behaviors of the system. Figure 2.2 shows an Kripke structure and its corresponding Büchi automaton, obtained following this transformation. It is worth noting that every state in this automaton is accepting, represented as double circles in the automaton.

An LTL formula representing a system specification can be transformed into an equivalent Büchi automaton [65][116]. Given the corresponding automaton $\mathcal{S}$, the language accepted by such an automaton, $\mathcal{L}(\mathcal{S})$, would be the set of accepted system behaviors.

The system $\mathcal{A}$ satisfies the specification $\mathcal{S}$ if

$$\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{S}), \tag{2.1}$$

i.e. the set of possible behaviors of the system is a subset of the behaviors allowed by the specification. Büchi automata are closed under intersection and complementation [34], i.e. there is an automaton which accepts exactly the intersection of the languages of two automata, and an automata that recognizes exactly the complement of the language of a given automata, respectively. This allows rewriting (2.1) as

$$\mathcal{L}(\mathcal{A}) \cap \overline{\mathcal{L}(\mathcal{S})} = \emptyset, \tag{2.2}$$

where $\overline{\mathcal{L}(\mathcal{S})}$ is the complement of the language $\mathcal{L}(\mathcal{S})$. If the intersection is empty, the system satisfies the specification. Otherwise, a counterexample can be provided.

This procedure is carried out in two steps: first the complement, and then the intersection. However, instead of computing the complement of the Büchi automaton

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

resulting from translating an LTL formula, it is easier to translate the complement of the formula itself. As for the intersection, given two automata $\mathcal{A}_1 = \langle \Sigma, S_1, T_1, S_{01}, F_1 \rangle$ and $\mathcal{A}_2 = \langle \Sigma, S_2, T_2, S_{02}, F_2 \rangle$, where $F_1 = S_1$, i.e. all states of $\mathcal{A}$ are accepting (like in the translation from Kripke structures to Büchi automata), the intersection is defined as follows:

$$\mathcal{A}_1 \cap \mathcal{A}_2 = \langle \Sigma, S_1 \times S_2, T', S_{01} \times S_{02}, S_1 \times F_2 \rangle, \tag{2.3}$$

where $(\langle r_i, q_j \rangle, a, \langle r_m, q_n \rangle) \in T'$ iff $\langle r_i, a, r_m \rangle \in T_1$ and $\langle q_j, a, q_n \rangle \in T_2$.

Checking for emptiness in a Büchi automaton is equivalent to checking for strongly connected components reachable from an initial state with accepting states in them. Given a automaton $\mathcal{A} = \langle \Sigma, S, T, S_0, F \rangle$, and a accepting run ρ, then ρ contains infinitely many accepting states from F. Since S is finite, there is some suffix ρ' of ρ where every state on it is repeated infinitely often. This means that every state in ρ' is reachable from another state in ρ', and therefore the states of ρ' are part of a strongly connected component. This component is reachable from an initial state, and contains an accepting state from F.

The search for such strongly connected components can be performed using a *double depth first search* (DDFS) algorithm [49], also called *nested depth first search* (NDFS) [80]. Figure 2.3 shows the NDFS algorithm from [80]. This algorithm relies on two global data structures: a state stack D and a set of visited states V. In both structures, each state is stored with a boolean value, called "nested". This value indicates whether the state was explored during the normal search or a nested search. The first stage of the depth first seach works as usual. When backtracking from an accepting state, however, a nested search is started. The global variable "nested" is set to true to indicate that this is a nested search, and "seed" is set to the accepting state. If "seed" is reached again from this nested search, then a counterexample can be reported.

It is possible to use this search algorithm generating the states of the automaton $\mathcal{A}$ only as needed. This technique is called "on-the-fly" model checking [49][59]. The automata $\mathcal{S}$ for the LTL property is generated completely and used to guide the generation of the intersection automata. Using this approach it is possible to avoid constructing the entire state space of $\mathcal{A}$ in many cases, either because the intersection rules with $\mathcal{S}$ prevents it, or because a counterexample is generated first.

2.1.4 Abstraction

One of most important techniques for dealing with the state explosion problem is abstraction. There are different abstraction techniques, but they share the goal of reducing the model to be analyzed, while preserving (some of) the properties of the original model. We will describe two abstraction techniques: the *cone of influence* and *data abstraction*.

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

```
Input: Automata A
Data: Stack D
Data: Statespace V
Data: State seed ← nil
Data: Boolean nested ← false
Procedure NDFS()
    Add_Statespace(V, A.S0, nested)
    Push_Stack(D, A.S0, nested)
    Search()
Procedure Search()
    (s, nested) ← Top_Stack(D)
    foreach (s,l,s') ∈ T do
        // Check if seed is reachable from itself
        if s' = seed or On_Stack(D, s', false) then
            Print_Stack(D)
            Pop_Stack(D)
            return
        else if not In_Statespace(V, s', nested) then
            Add_Statespace(V, s', nested)
            Push_Stack(D, s', nested)
            Search()
        end
    end
    if s ∈ A.F and nested = false then
        seed ← s                              // Reachable accepting state
        nested ← true
        Push_Stack(D, s, nested)
        Search()                              // Start nested search
        Pop_Stack(D)
        seed ← nil
        nested ← false
    end
```

Figure 2.3: Nested depth first search (NDFS) algorithm

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

The former focuses on reducing the size of the state transition graph, while the latter tries to minimize the size of each state.

The cone of influence technique [45], also called *slicing* [80], tries to reduce the size of the graph to be explored by identifying a subset of key variables, and discarding the rest and their associated operations. The cone of influence $C \subseteq V$ is constructed starting from a subset $V' \subseteq V$ of variables of the model, and iteratively selecting the variables that influence the value of that subset. The result is the minimal subset C such that $V' \subseteq C$, which contains all these variables. All the statements that operate on variables outside of C can be safely ignored. This transformation maintains the basic control flow of the model, e.g. no cycles are added or removed.

Usually, the initial subset V' is composed of the variables present in a formula to be checked. This reduces the model transitions to the bare essentials required for checking the satisfaction of that formula. It is worth noting, however, that not all types of correctness requirements can be checked on these reduced models, such as detecting deadlocks.

On the other hand, data abstraction [45][80][63] aims to reduce the possible values of the variables of the model, and thus the state space to be explored, such that meaningful analysis can still be carried out. For instance, an integer variable can be abstracted to only three values: negative, zero and positive. Usually, the control flow of models depend on simple relationships between their variables, such as a fork that depends on whether a variable is negative, zero or positive. By abstracting the values of these variables so that the control flow is maintained, the state space that needs to be explored can be greatly reduced.

Operations and formulas using these variables need to be adapted to use the new abstracted values as well. For instance, an integer variable x can be abstracted by its sign into two values, negative and non-negative, using a new boolean variable *neg_x* [80]. An assignment such as $x = 0$ would be translated into $neg_x = false$ and a condition such as $x > 5$ into $!neg_x$. Other operations result in more complex transformations, such as increasing x by one. If x is non-negative, then its value does not change. If x is negative, there are two outcomes: the operation results in a non-negative number or x remains negative. If the two alternatives are considered in the model via indeterminism, then this abstraction is said to be an *over-approximation* [52], as the abstract model contains more behaviors than the original model. It is also possible to construct an abstraction by *under-approximation*, i.e. one whose behavior is strictly less than that of the original model.

The assumptions that can be made on properties verified on the abstract model with regards to the original one depends on the type of approximation. In the case of over-approximation, if a property is verified on all possible traces of the abstract model, then it is also verified in all traces of the original model. With under-approximation, if a property violation is found, then it must also be present in the original model.

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

```
never  {    /* []<>p */
T0_init:
    if
    :: ((p)) -> goto accept_S9
    :: (1) -> goto T0_init
    fi;
accept_S9:
    if
    :: (1) -> goto T0_init
    fi;
}
```

Listing 2.1: Never claim automata for the LTL formula $\Box\Diamond p$

2.1.5 The Spin model checker

Spin [80] is a model checker that can be used to verify the correctness of concurrent software systems modeled using the Promela specification language. The focus of the tool is on the design and validation of computer protocols [79], although it has been applied to other fields such as controller verification [83] and synthesis [61].

Spin can check the occurrence of a property over all possible executions of the system specification. If a violation of the property is found, a counterexample is returned that makes it possible to trace the cause of the violation. If the user so desires, the model checking process can continue so more errors can be found in a single run.

There are several types of properties that can be checked with Spin, from simple state assertions to formulas written in LTL [102], as described in Section 2.1.2. Spin translates LTL formulas into never claim automata, which are a special type of Promela process which are executed after each system step, used to check the behavior of the system. It is also possible to write never claim automata by hand, to declare properties that would be hard to specify otherwise. Listing 2.1 shows an example never claim, obtained using Spin to translate the LTL formula $\Box\Diamond p$.

Figure 2.4 shows an overview of the tool's workflow for checking LTL properties. The Spin tool is used both to translate an LTL formula into a never claim automaton, and to generate the source code for the verifier given a Promela specification. This verifier carries out the model checking process, and returns a counterexample if an error is found.

The Promela modeling language

Promela (Process Meta-Language) is the language accepted by Spin for describing systems. It is not intended to be used as an implementation language, but rather as a specification language to describe systems at a high level of abstraction. Promela is an imperative-style language, with features such as processes, channels for inter-process communication and indeterminism.

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

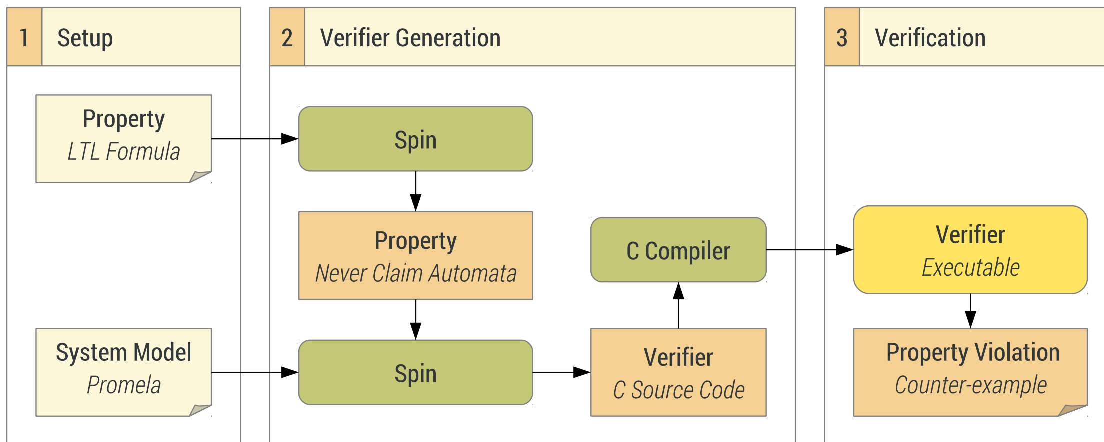


Figure 2.4: Overview of the Spin tool workflow for checking LTL properties

A Promela specification is composed of one or more process definitions (called `proctypes`), message channels and variables. Each process contains a sequence of statements, and can be started dynamically or at the beginning of the execution. The language provides built-in inter-process communication in the form of message channels. Channels can be either synchronous, i.e. rendezvous, or asynchronous, i.e. supported by a buffer of arbitrary size. Variables can be declared with a global scope, or locally withing a process declaration. Variables in the global scope are tracked by Spin and stored in the so-called state vector during the search. The supported data types are: `bool`, `byte`, `short` and `int`, as well as arrays of these elements and record structures. It is worth nothing that Promela does not have a floating point data type. Variables can hold symbolic variables using the special `mtype` type.

The control flow constructs include case selection (`if`) and repetition (`do`). These constructs allow one or more options, with an optional guard that dictates their eligibility. If no guard is active, the whole construct blocks. If several are at the same time, one of them is chosen non-deterministically. However, Spin's search algorithm will try to explore all possible indeterministic choices.

To avoid certain interleaving of process executions, sequences can be grouped in `atomic` blocks. These sequences are uninterruptable, and no other process can execute statements between the first and last statements of the block. However, if one of the sentences block, then the atomic sequence is broken and other processes may be executed. The `d_step` block is similar to `atomic` but with stricter rules, such as treating blocking statements as errors or resolving indeterminism in a deterministic way.

In order to provide some structuring mechanism similar to procedures, statements can be grouped inside `inline` definitions. `inlines` are more similar to macros than procedures: the text of the `inline` is substituted wherever it appears. `inlines` may have parameters, but they do not provide a separate scope.

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

```
#define MAX 100

mtype = { stop };

chan data = [1] of {int};
chan control = [0] of {mtype};

active proctype producer() {
    do
    :: data!1
    :: data!10
    :: control?_ -> break
    od
}

active proctype consumer() {
    byte msg, sum;
    do
    :: data?msg -> sum = sum + msg
    :: sum > MAX -> control!stop
    od
}
```

Listing 2.2: Example Promela specification

Listing 2.2 shows a small Promela example with two processes: `producer` and `consumer`. There are two channels used by these processes: `data`, an asynchronous channel of size 1 used to send integer values from `producer` to `consumer`, and `control`, a synchronous used to send control messages the other way around. `producer` may send either a 1 or a 10 value to `consumer`. When the sum of values sent to `producer` reaches a certain threshold, it may send an `stop` message back to instruct `consumer` to break from the loop and terminate. However, as it is written, `producer` may never send that message, as it always has the choice between that and accepting a new data message.

Embedded C code

One of the most powerful features of Promela is the ability to embed C code in the specifications. C code can be used inside of processes and as part of expressions. C variables can also be part of the global state and be used in temporal formulas. However, Spin does not check in any way the contents of embedded C code fragments.

The primary purpose of embedded C code is supporting automatic model extraction from C programs. However, it can be used to include external C code that may interact in some way with manually constructed models. The special built-in struct `now` contains the current values of the global variables, and can be read from or written to. For instance, in

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

this thesis we use embedded C code extensively to communicate with an external system, and make some of its internal state part of Spin's global state by modifying the contents of `now`.

Arbitrary C statements can be embedded using a `c_code` block. The statements in the block are executed atomically. `c_code` blocks can also be used to declare functions and global variables, or include external header files. The `c_expr` statements may contain C code that will be evaluated as an expression. If the body returns 0, the statement blocks, which is useful when used as guard in a control flow construct. The code inside a `c_expr` must be free from side effects, as Spin may have to evaluate it repeatedly. A `c_decl` block can be used in the gobal scope to declare new C data types.

Although global variables can be declared inside a `c_code` block, they remain outside Spin's state vector. However, there are two statements that allow putting C variables in the state vector. `c_state` statements can be used in the global scope to declare C variables that are global, local to any process, or hidden from the state vector. With the former, the variable is part of the state vector as any other regular Promela global variable. `c_track` can be used to include any external C variable or part of memory as part of the global state, and make it part of the state vector. This is useful for tracking C variables declared in external C code.

Listing 2.3 shows a simple example using some of the statements for embedding C code. First, a new struct type is declared using `c_decl`. This type is used in line 7 to declare a global variable called `d`. A C function is declared inside a `c_code` block with global scope, starting at line 10, so that it can be used elsewhere in the Promela specification. The global variable `d` is initialized inside a process local `c_code` block in line 18. Note how both the `d` variable and the `x` variable, which was declared as a regular Promela global variable, must be referenced using the `now` struct. Finally, the `do` construct contains a guard written as a C expression using `c_expr` in line 21. This expression calls the C function declared above, and must also use the `now` struct.

Search algorithm

Spin works by performing an exhaustive search of the model's state space, using a double depth-first search algorithm (see Section 2.1.3). First, the given LTL formula is translated into a Büchi automaton [115], represented as a never claim automaton, which is used to track the satisfaction of the formula. Then the synchronized product of the Promela specification and the automaton is explored using a double depth-first search algorithm. For each step of the system, a step of the never claim automaton is executed. Depending on the original formula, this automaton may have accepting states. If an acceptance cycle if found in the automaton during the search, or it finishes its execution, it means that a violation of the formula has been found, and a counterexample can be reported. On the other hand, if the automaton blocks because there are no executable instructions, it is not possible to find a violation of the formula following the current execution branch.

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

```
c_decl {
    typedef struct data {
        int a, b;
    } data;
}

c_state "data d" "Global"
int x;

c_code {
    int eq(int a, int b) {
        return a == b;
    }
}

active proctype example() {
    x = 10;
    c_code { now.d.a = now.x; now.d.b = 100; };

    do
    :: c_expr { eq(now.d.x, now.d.y) } -> c_code{ now.d.x += 10; }
    :: else -> break
    od
}
```

Listing 2.3: Example Promela specification with embedded C code

States visited along the current branch are stored in a *stack*, while another data structure, called *hash* or *heap*, stores every visited state to avoid exploring the same states twice. Figure 2.5 shows an overview of these data structures during a search.

2.2 Trace-based analysis

Our work can be compared to those of other analysis tools that treat the system under analysis as a black box, or which are concerned on analyzing traces generated during the execution. System models can also be analyzed in this way, but these techniques are specially useful for systems which are too large to analyze exhaustively, e.g. a real program, or one whose inner specification is not known.

We provide a more formal definition of execution traces with regards to our framework in Chapter 4, but it can be summed up and generalized as a sequence of discrete steps which reveal part of the state of the system in certain points during its execution. This information can be compiled from various sources, such as exposed system variables or aggregated information from external probes. In many cases, the analysis is only

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

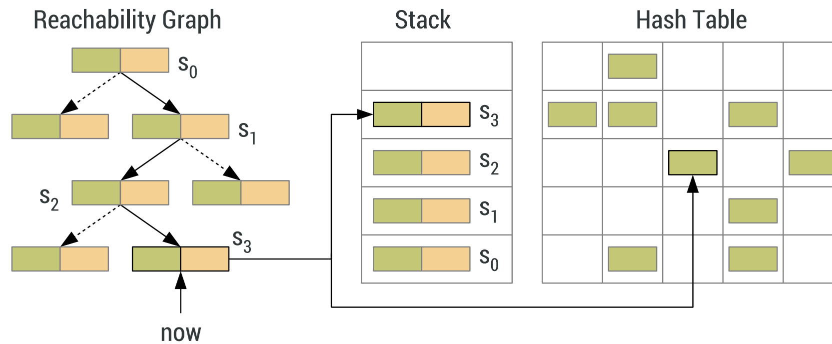


Figure 2.5: Data structures used by Spin

interested in a trace with "relevant" states, and "relevant" information per state. Some systems may be able to provide traces with only this information, while others may require some assistance before they can be consumed by the analyzer.

These tools are available for a wide range of systems and areas of application, from testing to performance evaluation. In the rest of this section we present some notable examples, whose approach is related to ours or whose area of application we have covered in this thesis. To enable a more meaningful comparison, we will organize these tools according to the applications presented in this thesis in Part III.

2.2.1 Analysis of network simulations

Communications networks and network software are two areas that have received significant attention from analysis tools. There are many approaches concerned with analyzing these systems during or after deployment, e.g. to check for performance issues on deployed networks. However, during the early development phases it is common to resort to simulation models in order to quickly iterate designs in a cost-effective manner.

There are a number of general-purpose network simulators, ranging from academic to commercial tools. Two of the most widely used in the academic community are the ns-2 network simulator [11], which is the subject of the applications described in Chapter 5, and its successor ns-3 [77]. Some of the tools which we will describe later take advantage of the capabilities offered by these simulators to perform their analyses. Commercial simulators such as OPNET [36] provide integrated tools for performance analysis.

Reliability analysis

Communication protocols have been widely studied from a reliability and correctness standpoint. Model checkers such as Spin [80] have been traditionally associated with this field [79], checking the correctness of protocol models. Other recent efforts have been

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

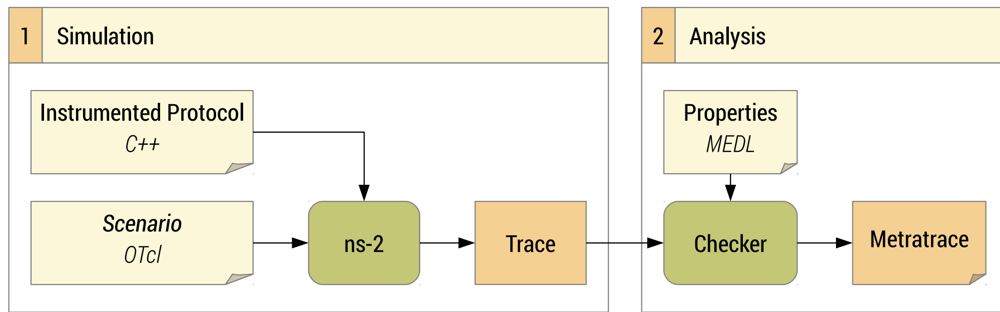


Figure 2.6: Overview of Verisim workflow

designed to deal with source code instead of models, such as CMC [95], Verisoft [68] and Java PathFinder [117]. In this section we will discuss other approaches that work with models for network simulators.

Verisim [32] is a tool that monitors ns-2 simulations and checks properties on the generated traces. Verisim requires the designer to declare the low level events that must be monitored, and the properties to be checked using MEDL (Meta Event Definition Language), which is an extension of LTL with auxiliary variables. Events are defined over the typical packet traces generated by ns-2. The result of the analysis is a *metatrace* that contains the violations of the specified properties in a single simulation trace. Figure 2.6 shows an overview of this workflow. However, unlike OptySim, Verisim performs the analysis off-line, after the whole trace has been generated. If simulations have to be stopped manually, the prefix of the trace given to Verisim could miss portions relevant to the properties being checked. In addition, Verisim does not provide support for generating and analyzing a large number of traces based on a given parameter space, requiring more manual interaction that our framework.

The Monitoring and Checking (MaC) module is at the core of the Verisim and, much like our work, has been adapted to analyze others systems such as Java [85]. The Java version introduces another language to specify the events that should be monitored, PEDL (Primitive Event Definition Language). The use of PEDL and MEDL can lead to a simpler definition of some complex properties.

Sobeih et al [111] extended the J-Sim network simulator [112] with a custom model checker. Models in J-Sim are written in Java as a set of loosely coupled components. Instead of checking properties on a trace, the extension performs an exhaustive search over the model's state space. To enable this search, the designer has to provide information on the composition of the global state of the model (to check the properties and backtrack to a previous state) and the events that can be triggered. The model checker can perform a best-first search if provided with an objective, which may lead to longer traces but in less time. However, the only properties that can be checked over J-Sim models with this model checker are state assertions.

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

Performance analysis

Parameterized simulations are common when evaluating a new protocol or tuning the parameters of an existing one for a given scenario. This kind of analysis has been supported by both commercial and academic tools. For instance, the OPNET Modeler[1] network simulator allows the definition of parametric studios, where several parameters will vary over a specified range. The statistics collected from each simulation can be plotted to show the results as a function of parameter values.

ANSWER [26], a tool for performing large-scale parameterized simulations with ns-2, presents a similar approach to our definition of parameterized analysis. An XML configuration file is used to declare the parameters that have to be combined to generate the set of different scenarios and its values, the metrics to be checked and other options. The *launcher* module generates the set of scenarios, performs the corresponding simulations in ns-2, and stores the output metrics in an organized way. A web application can then be used to plot diagrams for a single metric among a series of selected simulation scenarios.

Data collection and statistical analysis are facilitated by another tool [37]. This is complementary to our approach, since for our ns-2 integration we rely on the model designer to provide the relevant metrics. This framework also provides support for gathering statistics by performing a series of independent runs of the same scenario, e.g. until a confidence interval is reached, which our tool does not support.

In contrast with our approach, simulations are not checked against any requirement. In addition, OptySim provides an efficient mechanism to stop unproductive simulations on-the-fly. Combined with the exhaustive exploration of the parameter space, we can reach a greater coverage with the same resources, while providing results relevant to the user. Although the way in which ANSWER controls simulation campaigns increases the confidence and credibility of simulation results, the lack of mechanisms to check the correctness of the simulation model could leave design errors undiscovered.

Ye et. al [119] propose a simulator framework for the adaptive configuration of live networks according to a performance objective. Network topology and traffic patterns are monitorized and exported to a parameterized simulation setting using ns-2. Simulations are carried out and optimized using a black-box approach, where only the configuration parameters and performance metrics are known, employing a search algorithm to select new parameter values to simulate. Our approach also uses metrics from the simulations to control the process, but in an exhaustive manner. In addition, while Ye et. al use functions as the objective, our use of temporal formulas enables the definition of more complex objectives.

Automated simulations for analysis have also been used in [47] to evaluate the perceptual quality of VoIP calls. The use of simulations allows the authors to create

[1] OPNET Modeler was provided under the OPNET University Program.

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

repeatable tests in a controlled environment with different parameters, as opposed to regular call quality assessment, which relies on subjective tests and is more difficult to set up. However, like [26] and [119], no further steps are taken in order to guide the simulation process and reduce analysis time, as opposed to our approach.

There are also several works regarding the optimization of a protocol implementation, such as FIBER [84] or AEOS [58]. FIBER implements iterative methods to support auto-tuning of parameters at two different stages: installation and run-time invocation. AEOS can be used to optimize software automatically at runtime, using methods like search heuristics, and has been applied successfully to communication software problems like collective communications in MPI.

2.2.2 Analysis of Java programs

Programming languages themselves are the target of many tools, from development productivity tools to analyzers that help uncover errors in programs. Manual debugging, either with the help of a debugger or by simpler means, has traditionally been one of the most used techniques for discovering and fixing errors in programs. However, increasingly automatic methods such as unit testing or model checking are desirable to ensure that errors are discovered early and with minimal action from the developer.

Unit testing is a widely used technique for checking isolated parts of a program against a series of requirements, in a repeatable fashion. There are also model checkers aimed at analyzing real programs instead of models, targeting languages such as C or Java. In order to be effective, these tools should be well integrated with the development flow, and be usable by developers with little experience in their formal underpinnings. In this section we present some approaches to analyzing Java programs in these two categories.

Testing

Unit testing promotes writing self-contained, repeatable tests that check isolated parts of a system. Each test stimulates a certain part of the system with a series of inputs, and then compares the obtained and expected outputs. Software systems are usually composed of several components (from functions to libraries) which depend on each other. In order to test these components in isolation, it is often required to write *mocks* or *stubs*, i.e. implementations of other components oriented towards providing a controllable environment for the one being tested. Unit testing encourages testing components under extreme circumstances, such as corner cases, incorrect inputs, or errors from other components.

JUnit [21] is a popular unit testing framework for Java, originally developed by Kent Beck and Erich Gamma. It is based on SUnit, a testing framework for Smalltalk also developed by Beck, and has spawned multiple ports to other languages such as

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

C++ or Python. In JUnit, tests are written as class instance methods, and grouped in test cases and test suites. Earlier versions of JUnit relied on subclassing `TestCase` and naming conventions in order to define test methods, whiler newer versions rely on Java annotations. The framework also provides a set of helper assertion methods, to compare the expected and actual results.

TestNG [30] is another popular unit testing framework for Java, with similar functionality to that of JUnit. However, it provides more control for setting up tests, as well as better support for parameterized tests. Test methods may have parameters that can be provided in runtime from a parameter sources such as XML files, or data provider methods. The former are limited to a single parameter value per test, e.g. methods with multiple parameters and values require declaring an exponential number of declarations, one per combination. Data provider methods are more powerful, e.g. they allow using arbitray Java as parameters, but task the user with creating parameter combinations, if required.

Listing 2.4 shows part of a simple unit test for a fictional `Fibonacci` class, written with TestNG. Before each test method is executed, a new instance of `Fibonacci` is created as part of the test initialization. Expected exceptions within a test method can be declared in the `@Test` annotation. If the `testInvalidArg()` method exists without throwing a `IllegalArgumentException`, the test is marked as a failure. Finally, `testWithParams()` is a parameterized test, whose parameters are provided by the annotated `provider()` method.

Model checking

The most notable tools for analyzing Java programs using some variant of full-state model checking are Bandera [48] and Java Pathfinder [117]. Bandera is a tool based on model extraction that requires the Java program to be transformed into a model composed of pure Promela plus embedded C code. This model is optimized by applying a data abstraction mechanism that provides an approximation of the execution traces. As Bandera uses Spin as the model checker, it can check LTL on infinite traces and preserve correction results according to the approximation of the traces. Compared with Bandera, our Java integration only checks a set of traces. However the use of runtime monitoring to avoid model transformation, and our abstraction methods guarantee the correctness of the results.

Java Pathfinder [74][117] (JPF) is a model checker for Java programs which, in contrast with OptySim, performs a complete exploration of a program. In addition, thanks to its state matching mechanism, JPF does not revisit the same execution path twice, while OptySim analyzes each trace in isolation without checking for potentially repeated states between traces.

The JPF core only checks for "simple" defects such as deadlocks and unhandled exceptions. However, JPF can be extended to check for more meaningful problems

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

```java
public class FibonacciTest {
    private Fibonacci fibonacci;

    @BeforeMethod
    public void initialization() {
        fibonacci = new Fibonacci();
    }

    @Test
    public void test() {
        Assert.assertEquals(0, fibonacci.fibonacci(0));
        Assert.assertEquals(1, fibonacci.fibonacci(1));
        Assert.assertEquals(13, fibonacci.fibonacci(7));
    }

    @Test(expectedExceptions = IllegalArgumentException.class)
    public void testInvalidArg() {
        fibonacci.fibonacci(-1);
    }

    @Test(dataProvider = "testWithParamsProvider")
    public void testWithParams(int n, int expected) {
        Assert.assertEquals(expected, fibonacci.fibonacci(n));
    }

    @DataProvider(name = "testWithParamsProvider")
    public Object[][] provider() {
        return new Object[][] {{0, 0},{1, 1},{2, 1},{3, 2},{4, 3}};
    }
}
```

Listing 2.4: Example TestNG unit test for a Fibonacci class

through implementing *listeners* that monitor actions in the Java virtual machine (JVM). One such extension for checking LTL formulas with JPF is under development [90]. However, the program elements that can be referenced in LTL formulas are currently limited to method entry, e.g. $\Box\Diamond method()$ if `method()` should be called infinitely often, while OptySim allows a richer set of propositions. In addition, the user must declare explicitly whether the formula should be evaluated for infinite or finite traces.

The specification of LTL properties to analyze programming languages at runtime has been proposed by other authors, which we discuss in the rest of this section. Probably, the most complete overview of the approaches can be found in a paper by Bauer et. al [27]. Bauer et. al consider the runtime verification of LTL and tLTL (timed LTL) with a three-valued semantics (with truth values true, false, inconclusive) suitable to check whether or not a partial observation of a running system meets a property. They generate

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

deterministic monitors to decide the satisfaction (or not satisfaction) of a property as early as possible. They use these three-values as a way to adapt the semantics of LTL to the evaluation of finite traces. The authors write that "the set of monitorable properties does not only encompass the safety and cosafety properties but is strictly larger". However, the general case of liveness properties for infinite traces is not considered. Compared with our work, they develop the foundations to create monitors to support the new semantics of LTL for infinite traces, while our work relies on the already existing algorithms and tools to check Büchi automata for infinite traces.

Java PathExplorer, developed by Havelund and Roşu [75], uses the rewriting-logic based model checker Maude to check LTL on finite execution traces of Java programs. The authors provide different semantics for LTL formulas in order to avoid cycle detection. Java PathExplorer also supports the generation of a variant of Büchi automata for finite traces developed by Giannakopoulou and Havelund [66]. We share with Java PathExplorer the idea of using the model checker to process the stream of states produced by Java. However, our use of Spin allows us to check infinite execution traces.

The tool Temporal Rover [57] can check temporal logic assertions against reactive systems (with non-terminating loops) at runtime. The author considers that both finite and infinite traces are possible. However, only finite traces are evaluated, and a default *fail* value is returned for formulas like $\Diamond p$ when p has not been satisfied at the end of the trace and there is no evidence that the program has terminated. OptySim can provide a conclusive verdict when inspecting the infinite trace.

Bodden [33] uses AspectJ to implement a method to evaluate LTL, inserting pieces of Java code to be executed at points where the behavior specified by the formula is relevant and must be evaluated. This method is useful to check only safety properties. d'Amorim and Havelund [51] have developed the tool HAWK for the runtime verification of Java programs, which allows the definition of temporal properties with the logic EAGLE. In addition, the user must supply a method that must be called when the program terminates in order to produce a finite trace. FiLM (Finite LTL runtime Monitor) [120] also gives a specific semantics to LTL to check both safety and liveness in finite traces. However, in the case of liveness, manual inspection is required when the tool reports a potential liveness violation. All these tools for runtime monitoring of LTL are focused on finite traces. The main difference with OptySim is the support of cycle detection due to the way in which the states are abstracted and stored, and the use of Büchi automata.

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

Part II

The OptySim trace analysis framework

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

Chapter 3

Analysis framework

OptySim, our analysis framework, reuses the Spin model checker to analyze external systems instead of Promela models. These systems are analyzed by executing them and inspecting the resulting execution traces. These traces are sent to the model checker while the executions are still in progress, enabling the on-the-fly analysis of traces. Several traces are generated by exploring the parameter space within the established limits, and taking into account some of the external factors. The traces are checked for one or more *objectives* set by the user, that may be described using different formalisms. The interpretation of these objectives is open and can be used for several purposes, such as testing or optimization. These objectives also allow to stop the the executions as soon as a verdict is available for its trace, potentially saving considerable analysis time.

This chapter expands this description of our framework, including what inputs must the user supply, what outputs to expect, and how the analysis itself is performed. Additional focus is put into how traces are transmitted from the system under test and reconstructed in Spin using a generic communication protocol, and what kind of properties can be used as objectives for the analysis.

3.1 Framework overview

Figure 3.1 shows an overview of the OptySim framework, divided into steps. The following paragraphs walk through the framework at a high level, while individual steps are described in more detail in the following subsections.

First, in the setup phase, the user must supply both the system under test (SUT) and a configuration file describing the analysis to perform. This configuration file includes the declaration of the parameter space to be explored, the variables of interest of the SUT and the objectives to be checked on the generated traces.

In the second step, code generation, this configuration file and a framework template are used to generate a Promela file that contains the analysis algorithm, communication

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

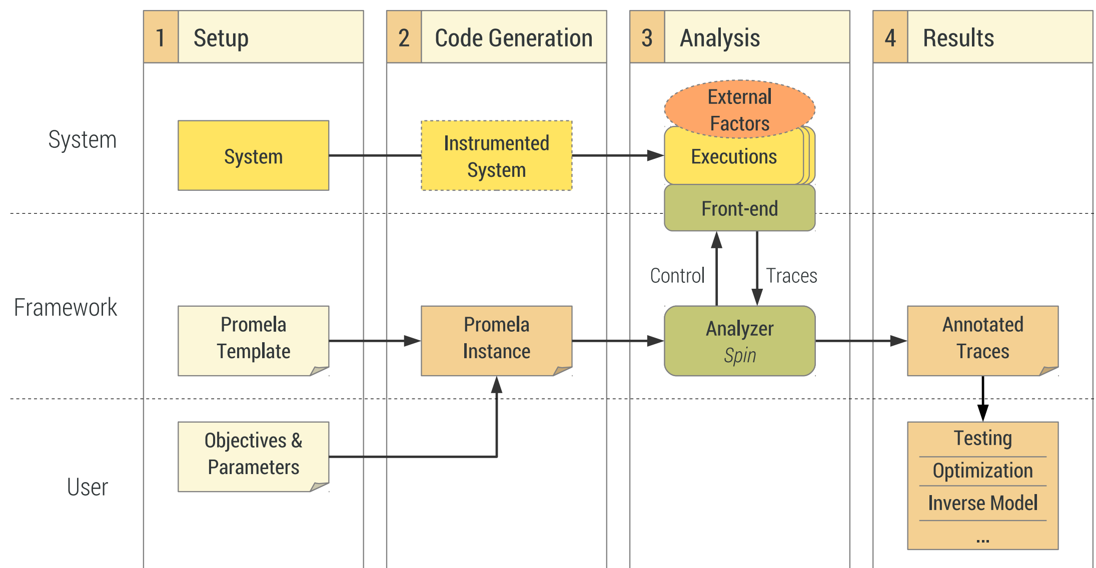


Figure 3.1: Overview of the OptySim analysis framework

routines and other required bits, tailored for this particular system and configuration. This Promela instance is processed by Spin to generate intermediate C code, which is then compiled into the executable analyzer.

The analysis happens in the next step, in which the executable analyzer explores part of the space state of the system. This is done in two dimensions. First, the parameter space is explored exhaustively according to the specification of the user. For each complete parameter configuration generated, the SUT is executed with this configuration. These executions generate execution traces, which are analyzed in isolation and on-the-fly by the analyzer, one at a time. The communication between the analyzer and the SUT happens through an established protocol. When an objective is met or violated in the current trace (or the trace ends), the analyzer generates and checks another configuration. This process continues iteratively until the state space is exhausted or the analyzer runs out of resources.

The raw result of this analysis is a series of execution traces, annotated by the analyzer according to the objectives. These annotated traces may be interpreted or used in a number of ways, such as test results or optimum parameters for some performance objective.

3.1.1 Setup

As the first step, the user must supply the system to be analyzed and the configuration to be applied to the analysis process. OptySim supports many types of systems, as long as they can produce an execution trace to be analyzed, either naturally or with additional

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

work from the user. For instance, network simulators such as ns-2 can be instructed to capture and show packet traces, but tracing the internal state of network components may require additional work.

The analysis configuration provides important information for carrying out the exploration and analysis of the system, written as an XML file. It can contain a combination of: parameter space declaration, analysis objectives, selected variables and additional options. In the rest of this section we describe these configurable aspects. The complete format of the XML configuration files is shown in Section A.1.

Parameter space

A single execution of the system produces a single execution trace. Many systems offer a series of configurable parameters which affect their behavior, while others are affected by external factors, both of which lead to different execution traces. In many types of analyses it is important to explore as many different execution traces as possible, or at least an relevant subset according to system parameters, external factors and user objectives.

The user can declare a parameter space which, when explored, will produce several parameter configurations for the system. For each system parameter, a combination of discrete values and value ranges can be specified. During the analysis phase, the analyzer will generate every possible combination of these values, producing a complete parameter configuration which will be applied to a new execution of the system.

Some external factors may be controlled to some extent, specially in the case of simulation models, and may appear as part of the parameter space. In other cases, the only automatic way which may yield different external conditions are multiple independent executions. For instance, randomness in a system may be controlled by a seed (an integer number) established at the start of the execution. Thus, the seed itself may be part of the parameter space, producing different random sequences in each execution.

Analysis objectives

The analysis step is guided by the objectives set by the user. The purpose of objectives is twofold. On the one hand, they are the main way of obtaining results from the analysis. On the other hand, they help reduce the state space that needs to be analyzed, either directly or indirectly, and thus the time required for the analysis.

Objectives may represent desired requirements of the execution traces, e.g. a video transmission between two peers is completed, or unwanted properties, e.g. the video transmission takes more that a limited amount of time. This leads to a broad classification of objectives into two groups, which we call *accept* and *reject*, respectively. When a trace complies with an objective, the trace is annotated as *accepted* or *rejected*, based on this classification of objectives.

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

These objectives may written using different formalisms of different complexity and purpose. Simple objectives may be written as state asserts, i.e. a boolean condition tested on each state independently. Complex objectives may be written using temporal logic, describing the evolution of the system over time.

There are other objectives that are used exclusively to limit the state space to be explored. For instance, certain combination of parameters can be excluded from being executed, e.g. using a boolean condition over the values of the parameters.

The kind of properties supported as objectives are described in more detail in Section 3.3 later in this chapter.

Selected variables

Our framework analyzes execution traces, which are sequence of system states, as they evolve during the execution. Usually, the full trace is not required to perform the analysis correctly. We abstract the original system trace in two dimensions, reducing the information contained in each state and the number of states of each trace. This abstraction is described in detail in Section 4.2, but is based around the idea of declaring a subset of the system variables. This subset is usually composed of the variables that appear in the declared objectives. The configuration file includes a section for declaring this subset of system variables.

Options

Finally, there are additional options that further refine how the analysis is carried out, or that are dependent on a particular type of system. These options are specified as key-value pairs. For instance, for ns-2 analysis, the “filename” option should contain the path to the main scenario file, and the packet trace for each simulation can be saved by setting the “tracePackets” options to *true*.

3.1.2 Code generation

OptySim reuses the Spin model checker, which typically analyzes Promela specifications. In our case, we want to analyze an execution trace from an external system. We use a special Promela specification which translates execution traces into Spin states on-the-fly. However, this Promela file requires an additional customization which depends on the user objectives and the variables contained in the execution traces. Thus, our framework offers a template which is then instantiated with the data given in the setup by the user.

This Promela template also includes a considerable amount of embedded C code, including calls to functions defined in a separate helper library. Following the usual workflow, we use Spin to generate C code from the instantiated Promela template, and compile and link it with other required libraries using a C compiler. The result is an

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

executable which contains Spin's analyzer algorithm and our own trace exploration and reconstruction algorithm.

Some systems may require an additional instrumentation step, to generate the required traces or customize their contents. The extent and complexity of this instrumentation step depends on the type of system. For instance, in some cases it can be transparent to the system. For the applications described in this thesis, we developed a series of software modules that could be used in ns-2 simulations, and Java programs. Further details of the Promela template, and these libraries and instrumentations can be found later in this chapter, in Section 3.4.

3.1.3 Analysis

The analysis step is the main step in the framework. This is were the state space of the system under test, in the form of execution traces, is explored to check the objectives set by the user. As mentioned before, the parameter space declared by the user is exhaustively explored, which leads to exploring a series of execution traces (one for each complete parameter configuration generated).

The analyzer and the SUT use an established protocol to transmit the execution trace, and to relay additional control commands. For instance, when the system is launched, there is a negotiation and configuration phase were the SUT informs the analyzer of the variables it can trace, and the analyzer selects the ones it is interested in. The SUT side of the protocol is managed by a thin front-end, which mediates between the system and the analyzer. The communication protocol is explained in more detail in Section 3.2.

In the analyzer side, the stream of states which conform a trace are incorporated into the model on-the-fly. Typically, Spin analyzes a Promela specification. This specification defines a finite graph that, when explored, results in all the possible executions. In our case, however, the Promela specification does not contain the model specification to create whole finite graph. Instead, the graph is constructed on-the-fly using the states received from the SUT. The core of our Promela specification is a loop which, through embedded C code, communicates with the front-end of the system to receive new system states, and turns them into new Spin states.

Figure 3.2 depicts a simplified version of the analyzer algorithm, which results from Spin executing the generated Promela file. Relevant parts of the Promela template are shown later in this chapter, in Listings 3.1 to 3.5 from Section 3.4. The outer loop generates all possible parameter configurations (see Listing 3.4), while the inner loop reconstructs the execution trace generated for a parameter configuration. The `nextState` function retrieves the following state from the execution trace, and updates the global state to reflect this new state (see Listing 3.2). Since the Spin exploration algorithm may backtrack at any given point, we also keep the current step in the trace in the global state, and a system state stack outside of the global state (not shown in the algorithm). The execution trace is stored in this stack during the analysis. If a new state is requested with

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

```
while !parameterSpaceExhausted do
    parameters ← generateParameterConfiguration()
    newExecution(parameters)
    step ← 0
    while !error and !finished do
        nextState(step)
        step ← step + 1
    end
end
```

Figure 3.2: Main loop of Spin executing the Promela template

a higher step than what is available in the stack, the analyzer must wait for a new state from the SUT. If Spin backtracks, the first states that the analyzer will request will most likely be already present in the stack (see Listing 3.5).

It is worth noting that this algorithm contains no logic for checking the objectives themselves. Although the final implementation contains additional code for handling and storing results, and some kinds of objectives have been implemented manually (e.g. checking the validity of a parameter configuration), Spin handles temporal logic (and related) objectives itself. The algorithm in Figure 3.2 shows how the execution graph that Spin is to analyze is constructed on-the-fly from information from the SUT, instead of being generated solely from a Promela specification. The Spin analysis algorithm, as explained in Chapter 2, remains unchanged and useful in this context.

This on-the-fly process of reconstructing and analyzing a system trace with Spin continues until an objective is verified or violated, or the trace ends (either naturally or due to an error). Then, the analyzer generates another complete parameter configuration, and analyzes the resulting execution trace. This process continues iteratively until the state space is exhausted or the analyzer runs out of resources. However, the analyzer can also terminate after the first time an objective is checked, which can be useful for certain kinds of analysis.

3.1.4 Results

Raw results are local to each generated and analyzed execution trace. Each trace is fully dumped and annotated with a result assigned to that particular trace. The possible results mirror the types of properties that can be used, as explained in Sections 3.1.1 and 3.3. That is, for each type of objective, there is a result annotation which declares that the given objective was met. There are also additional result annotations for other purposes, such as an error in the system during the execution, or simply that the trace was completely generated and inspected without meeting any objective.

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

The full list of possible annotation results is as follows:

- ***accepted.*** The objective that yielded the result was an *accept* one.
- ***rejected.*** Same as above, but when the objective is a *reject* one.
- ***finished.*** The execution trace was finite and the analyzer reached its end before any result could be reached.
- ***error.*** There was an error during the execution of the system, before any result could be reached.

However, these results should be interpreted by the user according to the intended use. As explained before, OptySim can be used for a range of purposes, providing that generating and analyzing a range of execution traces is useful.

For instance, a straightforward use would be system testing. An objective may represent an error condition that the system should not reach. If a trace reached such condition, it would be stopped and annotated as *rejected*. The information contained in the trace, e.g. relevant variable values and the program counter, would be useful to locate the cause of the error. The parameter space would be declared so as to get as much coverage for the relevant portion of the system as possible.

Another use would be parameter optimization according to a some given objective. The system under test exposes a series of parameters that can be tuned, e.g. in order to attain a performance objective. In this situation, the parameter space would describe the possible values of these system parameters. The final result would be one (or more) complete parameter configurations that make the system reach the stated goal, i.e. a system optimized according to the given objective. In some cases, such as simulation of a real system influenced by external factors, the parameter space may also include these factors. Here the results would be a set of parameter configurations, each associated with some particular external conditions (as set in the simulation scenario). For instance, the proper configuration of a network protocol in a mobile environment may depend on the particular conditions of the environment in a particular moment, and the protocol should adapt, i.e. apply the optimum configuration, for each situation.

The applications in Part III include different uses of the results, including testing, performance optimization and modeling.

3.2 Protocol

This section describes the protocol used to communicate the analyzer and the system under test. It is worth noting that this protocol is independent of the particular analysis algorithm used, so it can be reused for other frameworks that perform analysis of execution traces. Supporting another type of system in OptySim is a matter of implementing

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

the protocol, e.g. as front-end for the system, and connecting it to the analyzer. The protocol can be divided into three phases:

- **Negotiation.** The first phase deals with extensions to the analyzer, e.g. options particular to some kinds of system. The SUT may declare a series of features it supports, which may alter the way the analyzer works or prompt it to make a choice among the offered features.
- **Setup.** Once both sides have agreed upon a series of features and options, the analyzer can choose which variables and events will appear in the execution trace (among those offered by the SUT), and may set the values of the system parameters before the execution starts.
- **Execution.** This is the main phase of the protocol. The analyzer will command the SUT to start its execution, which will lead to the generation and transmission of the execution trace. Each state of the trace will be sent separately. At any moment, the analyzer can command the SUT to stop, e.g. because a result has already been found. In turn, the SUT can inform the analyzer of non-trace events such as errors or system termination.

We defined a protocol that is useful for these phases and supports a certain amount of extension, but is simple enough. The remainder of this section is devoted to describing the protocol and its use. Appendix B includes a Promela model of the protocol and the complete definition of the protocol messages, using Google Protocol Buffers [14].

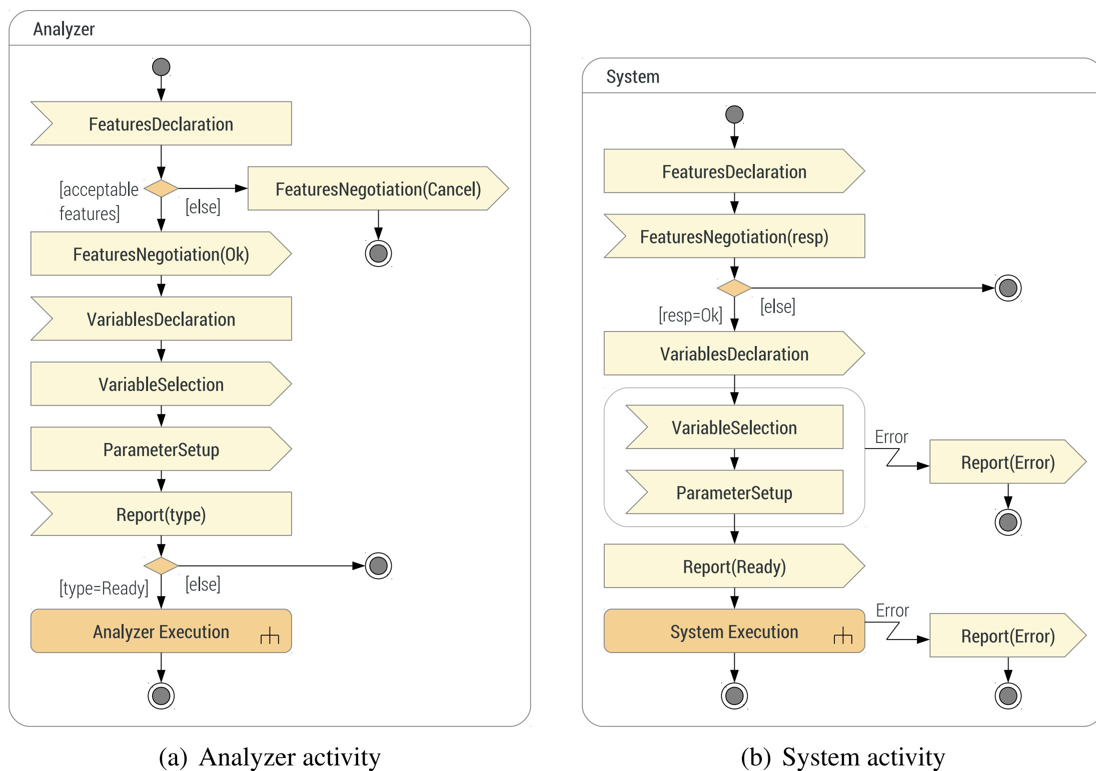
Figures 3.3 and 3.4 show an overview of the two sides of the communication protocol, as activity diagrams. Most message parameters have been omitted from these and following diagrams for simplicity. The only parameters included are those relevant to the evolution of the protocol. Also, these diagrams are deliberately devoid of any actions beyond sending and receiving messages (with a single exception).

3.2.1 Negotiation

The first phase of the protocol is the negotiation phase. In this phase the system declares a series of features, options or requirements, and the analyzer must decide whether it supports them, and select among the given options if needed. This is done with two messages: a *FeaturesDeclaration* sent from the system and a *FeaturesNegotiation* as a response.

The first message carries a series of features that the system declares to the analyzer. Each feature includes a name and may include one or more values. In addition, features can be flagged as *required* and *selection*. The former indicates that the feature cannot be ignored and must be explicitly supported by the analyzer, while the latter indicates that the analyzer can select one of the values associated to the feature. It is worth noting

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA



(a) Analyzer activity (b) System activity

Figure 3.3: Communication protocol overview: main activities

that *selection* does not implicitly include *required*. If the first flag is present, but not the second, it is inferred that the selection does not affect the way in which the analyzer must behave, even if it may result in a different behavior on the system side and even produce different execution traces. The analyzer may choose randomly or, better, let the system choose the default value.

The analyzer responds with a *FeaturesNegotiation*. This message includes, at least, a response field indicating whether the analyzer supports the required features (*Ok*) or not (*Cancel*). In the second case, the message exchange between both parts would end there without further negotiation, as it is understood that the system cannot be analyzed with the current analyzer, and the system would terminate. In the first case, the message can also include a selected value for some of the features which were declared with the *selection* flag.

The negotiation phase serves as a lightweight extension point. However, we have defined a few generic features, which should be recognized by the analyzer and may be used by any system:

- **One-time negotiation.** The negotiation phase can be skipped after the first time, if the analyzer so chooses it.

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

- **One-time variable setup.** The variable declaration and selection can be skipped after the first time, if the analyzer so chooses it.
- **Skip variables declaration.** The SUT will not send a *VariablesDeclaration* message. In this case, the *VariableSelection* message should contain enough information for the system to recognize the selected variables, e.g. by name. This is usually desired if the number of traceable system variables is too high and/or the analyzer already knows which variables it needs to trace without input from the system.
- **Reset support.** The SUT supports the *Reset* command, which should decrease the overhead when launching several executions in succession.
- **On-demand reports.** The SUT supports the *Report* command, which instructs it to submit a report with the current state of the execution.
- **System type.** A single feature which stands in for a set of features which are specific to a particular type of system, which the analyzer should support explicitly.

Some combinations of these features, such as one-time negotiation, one-time variable setup and reset support are traits of systems whose front-ends are persistent, i.e. they stand on-line and retain state information in between executions.

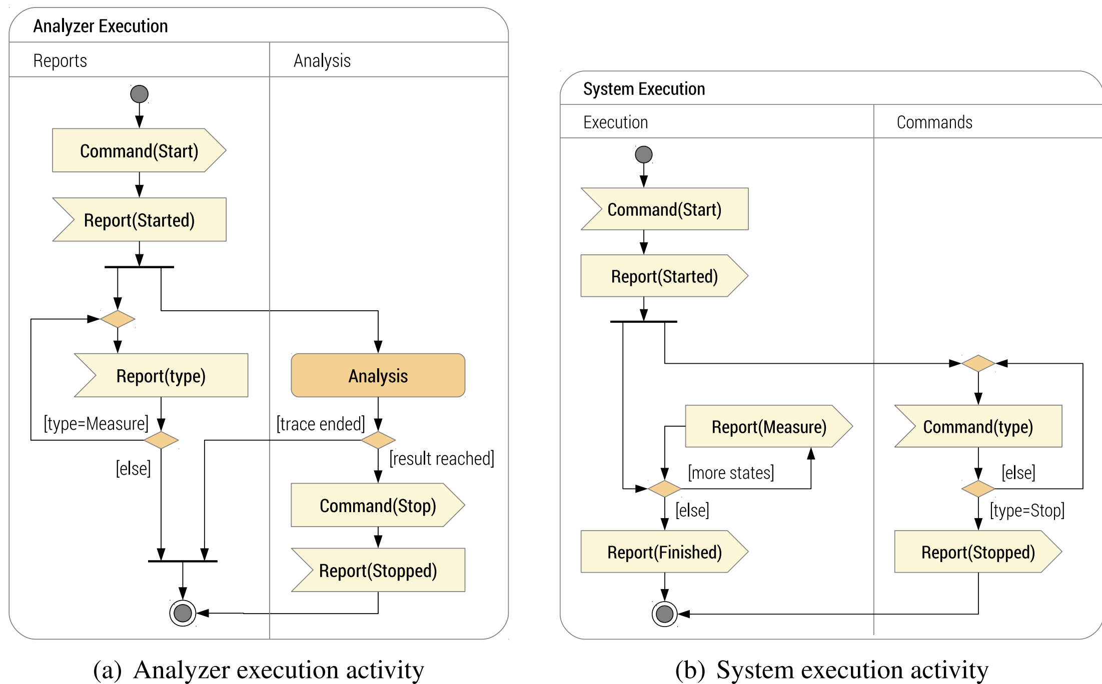
It is worth noting that supporting a feature does not mean that it will be taken advantage of, unless it is marked as *required*. For instance, if a system supports reset, the analyzer may still stop and start systems as if that feature had not been declared. Hence, in such cases the system should support both cases.

3.2.2 Setup

In the next phase, the analyzer selects the variables it wants to appear on the trace and sets the values of the system parameters. This phase is initiated by a *VariablesDeclaration* message from the system, which declares the available variables for monitoring. For each variable a numeric identifier is provided and, optionally, its name, type and whether its value can be set as a parameter before the system starts. Some of these variables may have a special significance depending on the type of system and the negotiated features.

The analyzer then selects the variables that are required for the trace from the set of declared variables, sending a *VariableSelection* message. This message may also assign new identifiers to the selected variables, to accommodate the expectations of the analyzer. In the rest of the protocol variables will be referred by this identifier and never by their names. Then, the analyzer prepares a *ParameterSetup* message with the values for the system parameters. Upon receiving it, the system shall set these values before the execution starts.

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA



(a) Analyzer execution activity

(b) System execution activity

Figure 3.4: Communication protocol overview: execution subactivities

Finally, the system acknowledges that the setup phase is completed and that it is ready to execute by sending a *Report* with a *Ready* type field. However, if the system detects an error in any of the responses sent by the analyzer, it may respond with a *Report(Error)* and terminate.

3.2.3 Execution

The execution phase of the protocol is shown in the subactivities of Figure 3.4, which are called at the end of the activities in Figure 3.3. In both sides the activities are partitioned into two swim lanes, representing parallel action paths. Note that some error conditions and commands have not been included in the figures for the sake of simplicity.

In the case of the system (Figure 3.4(b)), on the one hand it must send the relevant state information to the analyzer, and on the other hand it must watch for incoming commands from the analyzer. For the former, a *Report* message of type *Measure* is sent for each relevant state, packing the values of the variables at that moment. This may continue until the execution terminates normally, which shall be notified to the analyzer sending a *Report(Finished)* message before terminating. At any time, a *Command* message may be received from the analyzer, which should be handled as soon as possible. These commands include *Measure*, which forces the analyzer to send a new *Report(Measure)* with the current state, and *Stop*, which instructs the system to terminate its execution. In

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

this case, the analyzer is notified with a *Report(Stopped)* before the system terminates.

The analyzer activity is also split into two (Figure 3.4(a)), with one side devoted to receiving *Report* messages from the system, and the other one to the analysis itself. Here we have included an "Analysis" action, which contains the whole process of on-the-fly analysis as carried out by Spin with the reconstructed trace. If the analysis reaches any conclusion before the trace ends, the execution of the system is stopped by sending a *Command(Stop)* message and waiting for the confirmation in a *Report(Stopped)* message. Meanwhile, the analyzer may receive *Report* messages of several kinds. Several *Measure* ones may be received with new trace states. Any trace state data is stored in a state stack which is shared with the "Analysis" action. If a *Report* of type *Finished* or *Error* is received, then the analyzer assumes that the system has terminated and that no more states are going to be added to the reconstructed trace. Note that the analysis may continue even after the system has stopped, as the execution trace may not have been completely analyzed, or there may be branches that are still unexplored. The analyzer can terminate this phase as soon as it exhausts the trace and branches.

3.2.4 Example

Figure 3.5 shows an example scenario of the communication protocol in use as a UML sequence diagram, with two alternative outcomes. This diagram shows only one execution of the system. As with the activity diagrams in this section, most of the message parameters have been omitted. The example also shows which messages correspond to each of the three phases.

The negotiation and setup phases are carried out normally. In the execution phase, the system under test starts sending a series of *Report(Measure)* messages, with the states of the current execution trace. The first alternative shows how the analyzer would stop the execution of the system, typically after an objective has been met in the current trace, sending a *Command(Stop)* message. The SUT acknowledges this with a *Report(Stopped)* message. In the second alternative, the SUT finishes its execution naturally and notifies the analyzer of this event with a *Report(Finished)* message. The analyzer does not have to respond to this message. In both cases, the system terminates after sending its last message, and the analyzer may continue exploring another execution.

3.3 Objectives

Objectives may be written using several formalisms, as long as they can be translated into a never claim automata for Spin (see Section 2.1.5). In our framework we have added a couple of additional kinds of objectives to improve its usefulness. For instance, we allow the definition of asserts that are checked independently of any specified never claim automata. In addition, we have added a simple classification for the results, which

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

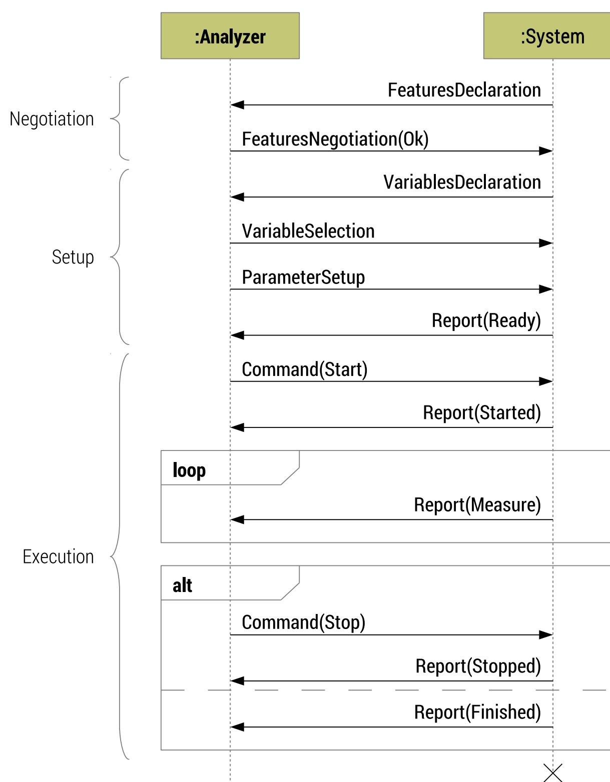


Figure 3.5: Example scenario of the communication protocol

make them easier to understand for the user. A summary of the supported objectives can be found in Table 3.1.

Most objectives include either *accept* or *reject* as part of their names, so that the user can decide whether a trace that verifies the objective should be tagged as *accepted* or *rejected*. For assertions, if the condition is true in some state then the trace is tagged. For never claim automata and related objectives, if a violation is found then the trace is tagged. If several objectives are specified, the trace will be tagged according to the first objective met and the analysis of that trace will be stopped. Note that if, for instance, a trace does not satisfy an *accept* objective, it does not mean that is should be tagged as *rejected*. However, we have found that it is often a good practice to set bounds for the analysis of traces using tagged objectives. We will discuss other available search optimizations in the next section.

We allow three main variations of these *accept* and *reject* objectives, summarized as follows:

- *accept*, *reject*: a simple state condition (assertion) over the variables of the execution trace

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

Objective	Parameter	Summary
def	Boolean formula or constant	Define a formula or constant as a symbol to be used in other objectives
accept *reject*	Boolean formula	Stop & tag trace if a state checks the given assertion
ltlAccept *ltlReject*	LTL formula	Stop & tag trace if the trace checks the given formula
neverAccept *neverReject*	Never claim automaton	Stop & tag trace if the given automaton reaches an accepting state
invalidConfig	Boolean formula[2]	Skip execution if parameter configuration checks the given formula
acceptSimilar *rejectSimilar*	Boolean formula[1]	Skip execution & tag if trace with "similar" configuration and same tag exists

Table 3.1: Summary of supported objectives.

- *ltlAccept*, *ltlReject*: an LTL formula which uses the variables of the execution trace
- *neverAccept*, *neverReject*: a never claim automata to be used directly by Spin, which uses the variables of the execution trace

Only one *ltl** or *never** objective may be used at the same time. However, we allow the use of several simple *accept* and *reject* objectives in the same analysis. For instance, the user may specify a desired evolution of the variables of a network model as an *ltlAccept* objective, while setting a few assertions as *reject* objectives to check for bad states and limit the state space, such as in the following example:

$$\begin{aligned}
def: &\quad start \;:=\; var = 1\\
def: &\quad maxTime \;:=\; 1000\\
ltlAccept: &\quad \Diamond(start \;\wedge\; \Diamond(var = 2))\\
reject: &\quad (var < 0 \;\vee\; var > 2)\\
reject: &\quad time > maxTime
\end{aligned} \tag{3.1}$$

In Spin, LTL formulas are usually written using boolean propositions defined elsewhere, such as *start* in the previous example, although variables can be used directly.

[1]Over the parameters of the current configuration and a previous one
[2]Over the parameters of the current configuration

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

This is the purpose of *def*, i.e. to define a boolean formula or a constant as a symbol that can be used in other objectives. In this example, the first objective accepts traces where the value of *var* will be 1, and then 2 at some point in the future. At the same time, traces where *var* is not in the [0, 2] range or where *time* exceeds 1000 are tagged as *rejected*.

In the following subsections we expand the description of these objectives and give some examples of each. Here we will use mathematical notation for the examples (including symbols such as $\Diamond$ or $\wedge$), but it is worth noting some conventions used when writing these formulas in analysis specification files:

- boolean operators are written as in C, e.g. `&&`, `||`,
- temporal operators are written as in Spin, e.g. `[]`, `<>`,
- variable names are prefixed with a dollar sign, e.g. `$var`

3.3.1 State assertions

State assertions, defined using either *accept* or *reject*, are the simplest kind of objective. A state assertion is a boolean formula over the variables of the execution trace, and is checked after a new state is explored in the analysis algorithm. If the formula is true given the current values of the variables, the analysis of that trace is stopped, the trace is tagged with either *accepted* or *rejected* accordingly, and the analysis proceeds to the next trace.

Some examples of simple state assertions include:

$$accept: \quad playCount > 1000 \tag{3.2}$$

$$reject: \quad time > 100 \vee rebufCount > 3 \tag{3.3}$$

Objective 3.2 simply checks that a system variable, *playcount*, reaches a certain value, while objective 3.3 requires either of two variables to reach certain limits. If these two objectives were used at the same time, the trace would be stopped as soon as one of them were true on a state of the execution trace, and tagged with *accepted* or *rejected*, respectively.

3.3.2 LTL formulas and never claim automata

Complex objectives that deal with the evolution of variables along an execution trace can be expressed usig LTL formulas or never claim automata. The syntax and semantics of LTL formulas are discussed in Section 2.1.2. As per the model checking algorithm used by Spin and discussed in the same section, LTL formulas are translated into Büchi automata and then represented as never claim automata. It is easier to check the presence

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

of a behavior (only one counterexample must be found) than the absence of one (the whole state space must be explored). Therefore, desired objectives written as LTL formulas are usually negated before translating them into a never claim automata to find violations.

Instead of providing a formula for translation, the user may provide a never claim automaton directly, e.g. obtained using another translation algorithm different from the one used by Spin.

At most one of these objectives is allowed at the same time, e.g. a single LTL formula or a single never claim automaton.

We now provide some examples of LTL formulas used as objectives:

$$ltlAccept: \quad \Diamond(var = 1 \wedge \Diamond(var = 2)) \tag{3.4}$$

Objective 3.4 describes the evolution of the variable *var*: for the trace to be "accepted", *var* must be equals to 1 at some point, and then equals to 2 later in the trace. The formula makes no assumptions on the value of *var* between these two points in the trace.

$$ltlReject: \quad \Diamond\Box broken \tag{3.5}$$

On the other hand, objective 3.5 will "reject" traces where *broken* is always true, starting at some point of the execution trace. *broken* may be a boolean variable or a symbol defined elsewhere, such as a boolean formula. This formula can only be checked on infinite traces, where *broken* is infinitely repeated. Spin stuttering mechanism, which transforms a finite trace into an infinite one by considering that the last state is repeated forever, can be used to check this objective on finite traces, e.g. if *broken* is true on the last state of the trace.

3.3.3 Parameter configuration validation

Valid parameter values can be defined as discrete values, ranges, or a combination of both. However, specially in systems with a significant number of parameters, some combinations of these parameters may not make sense, or are not interesting from the point of view of the analysis. To reduce the number of parameter configurations that have to be executed and analyzed, the user may provide additional objectives that validate these configurations.

There is a single objective of this kind, called *invalidConfig*, which accepts a boolean formula over the parameters of the system. Any number of *invalidConfig* objectives may be used at the same time. These objectives are evaluated after a complete parameter configuration is generated. If one of the is true, then the parameter configuration is tagged as *invalidConfig*, and the system is not executed and analyzed with these parameters.

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

For instance, objective

$$invalidConfig: \quad (p1 > 5) \wedge (p2 < -10) \tag{3.6}$$

would discard parameter configurations where the given formula is true.

3.3.4 Inferring results from previous analyses

In some circumstances, the result expected by executing the system with a given parameter configuration can be inferred from the result obtained from other similar parameter configurations. Identifying and acting upon these similar configurations may yield significant analysis time gains, wherever possible.

In the OptySim implementation, parameter configurations and their corresponding execution traces are explored sequentially, i.e. a new trace is generated and analyzed only after the analyzer has finished with the previous one. The results obtained from the analysis of previous traces can be applied to infer the result of the next trace, based on the objectives being checked, the generated parameter configurations and the known behavior of the model.

We provide *acceptSimilar* and *rejectSimilar* objectives to take advantage of this optimization. These objectives take a boolean formula over the parameter values from the current configuration and from another previous configurations. After a new parameter configuration is generated, the formula is evaluated for each previous configuration of with the same tag, i.e. *accepted* and *rejected* for *acceptSimilar* and *rejectSimilar*, respectively. If the formula is true for any previous parameter configuration, then the system is not executed with this parameter configuration, but it is tagged as if it had been (with no execution trace attached to the parameter configuration).

Definition 3.1 (Result inferred from previous analyses) *Given* $p = \langle p_1, p_2, ..., p_n \rangle$ *the values of the current parameter configuration. Assume that* Q *is the set of all previous parameter configurations, and* $q = \langle q_1, q_2, ..., q_n \rangle \in Q$ *the values of a previous parameter configuration. Let* $Tag = \{\text{accepted}, \text{rejected}, ...\}$ *be the set of possible tags. Let* $tag(q) : Tag$ *be the tag given to the trace that resulted from executing the system using* q*. Let* $obj : \{\text{acceptSimilar}, \text{rejectSimilar}\}$ *be the objective, and* $eval(obj, p, q) : Boolean$ *the function that evaluates the formula associated with obj using p and q as the values of the current and previous parameter configuration, respectively. Let* $tag(obj) : Tag$ *be the tag associated with the given objective, such that* $tag(\text{acceptSimilar}) = \text{accepted}$ *and* $tag(\text{rejectSimilar}) = \text{rejected}$*. If*

$$\bigvee_{q \in Q} ((tag(obj) = tag(q)) \;\wedge\; eval(obj, p, q)) \tag{3.7}$$

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

is true, then the system is not executed using the parameter configuration p, and the (empty) trace is tagged with $tag(obj)$.

Note that formula 3.7 can be evaluated in short circuit. That is, it is enough to find one $q \in Q$ for wich the formula is true.

For instance, given $p = p_1, p_2, ..., p_n$, if there exists a previous *accepted* parameter configuration with a smaller value for parameters p_1 and p_2, while the rest of the parameters have the same values, the user may declare that the next trace should be *accepted* as well without having to execute the model:

$$acceptSimilar: \quad (p_1 > q_1) \wedge (p_2 > q_2) \wedge (\bigwedge_{2<i\leq n} p_i = q_i) \tag{3.8}$$

In the previous objective, p_1 and p_2 refer to the values of these parameters in the current configuration, while q_1 and q_2 refer to the values from a previous configuration, i.e. from a parameter configuration $q \in Q$. The last portion of the formula states that the rest of the parameters in the configuration must be equals between the current and the previous configuration. When writing the objectives in test specification files, `$*` can be used as a short hand notation for this. Also, the variables of a previous configuration would be written as usual, but without the dollar sign.

However, special care must be taken when using this exploration optimization. If the relationship used for inferring results is not correct, the analysis will most likely provide false results.

3.4 Implementation

This section deals with the implementation of OptySim with respect to the analyzer. The starting point is an analysis specification file, provided by the user, and a Promela template with associated C libraries, provided by the framework. In the first step, the template is instantiated using the information provided by the user, and compiled into an analyzer executable. This analyzer is responsible for controlling and communicating with the system executions which yield the traces to be analyzed.

We will describe the main contents of the Promela template and how it is instantiated. We will discuss how the analysis algorithm of Chapter 3.1.3 is realized in the template, and what data structures are used.

3.4.1 Promela template

A template was necessary for our purposes as Promela state variables cannot be declared dynamically. State variables are declared with a global scope, outside any `proctype` or

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

`inline` definition, but our framework must deal with systems which expose different variables in the execution traces. Thus the declaration of these state variables must be tailored for a specific system.

The Promela template contains the relevant algorithms and data structures, but with a few placeholders that should be filled before it can be used. These placeholders include the aforementioned state variables, and the never claim automaton and additional boolean conditions, if any. These (and other relevant information) must be provided by the user in an analysis specification file, as explained in Section 3.1.1. The instantiated Promela template is ready to be processed by Spin into compilable C code.

We used the Apache Velocity template engine [12] to define and instantiate the Promela template. Velocity is Java-based: it can be used programmatically as a Java library, and Java objects can be passed to and used within the template. The Velocity Template Language (VTL) is simple but powerful. Most notably, arbitrary Java methods can be called on objects passed to the template. In VTL, variables are referenced with a leading dollar sign (e.g. `$message`), fields and methods on objects are accessed using the familiar dot operator (e.g. `$message.Sender`, `$message.setResult(true)`) and directives (such as conditions and loops) are preceded by a hash (e.g. `#foreach( $message in $messageList )`).

Figure 3.6 shows an overview of the contents of the Promela template. The template itself is divided into several files, and there is additional functionality provided in a separate C library. The template contains the declaration of the system variables that should be stored in Spin's state vector, and additional data structures required for the analysis and communicating with the system. It also provides placeholders for the different objectives: a never claim automaton (either provided directly or translated from another formalism), state assertion formulas, and checks on the generated parameter configurations. Parameter configuration generation is handled taking advantage of Spin's search algorithm and indeterminism. The values and ranges of these parameters must be instantiated on the template as well. The template is also responsible for launching a new system execution and managing the communication with it, so that execution traces can be reconstructed in Spin.

3.4.2 Data structures

The Promela template and related C libraries contain several data structures required to hold the state of the system execution, support backtracking and handle the communication with the system. This section covers the most important of these data structures and their use within the analyzer.

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

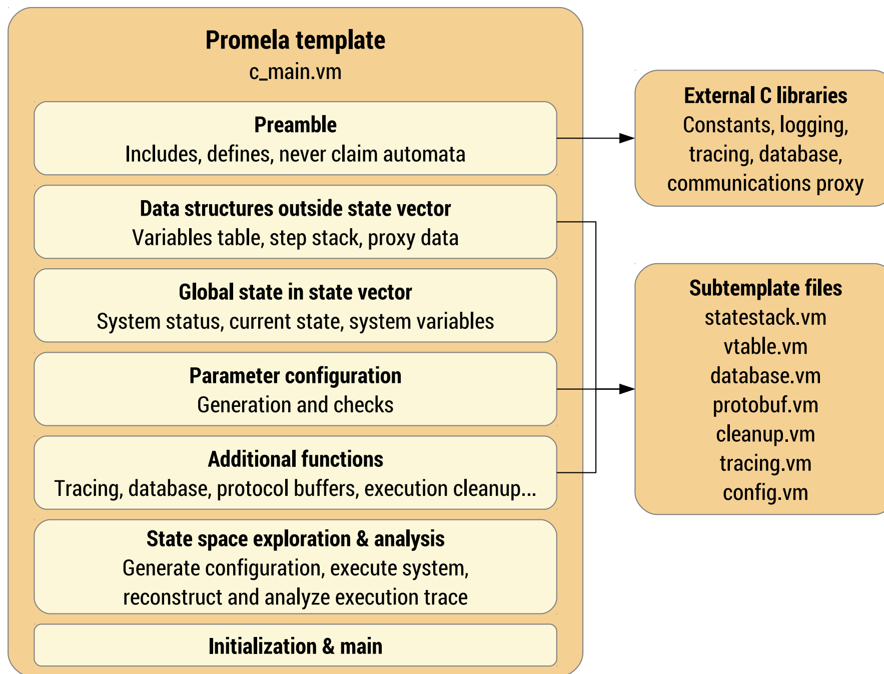


Figure 3.6: Overview of the Promela template contents

Data inside Spin's state vector

Part of the data used by the analyzer is inside Spin's state vector. As explained in Section 2.1.5, these variables will be stored in the stack and in the heap, and will be restored correctly in the event of backtracking. Most importantly, the states of the execution trace must be stored here so they can be inspected by Spin's analysis algorithm.

Listing 3.1 shows the part of the global state that is stored inside Spin's state vector. The first two variables shown, `running` and `currentState` are used to keep track of the state of the system and its execution trace. `running` declares whether the system is currently under execution or not. More precisely, it marks the end of an execution trace (if it is finite and it has been reached). `currentState` contains the index of the current state in the execution trace being analyzed. This variable is critical for supporting backtracking, in conjunction with the state stack, which is discussed in later in this section. Finally, system variables are declared using `c_state` as global variables. This enables using types commonly found in systems but not supported in Promela, such as floating point numbers. However, care must be taken when using such variables in formulas due to the nature of their representation.

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

```
bool running = false;
int currentState = 0;

#foreach( $parameter in $allVariablesList)
c_state "$parameter.CType $parameter.Name" "Global"
#end
```

Listing 3.1: Promela template: global state

```
struct state {
#foreach( $parameter in $allVariablesList)
    $parameter.CType $parameter.Name;
#end
};
```

Listing 3.2: Promela template: struct state for state stack

State stack

In the previous section, it was shown how the current state of the execution is stored in global variables declared with `c_state`, to be kept inside Spin's state vector. The values of these variables are restored in the event of backtracking. Execution traces are linear by definition, but never claim automata may induce branching in the analysis of a single trace. As a depth-first search algorithm is used for state space exploration, this may lead to backtracking while analyzing a trace. However, when advancing from a backtracked point, the trace reconstruction algorithm should not accept new states from the execution, but rather restore the already visited states (while they are available).

To support this, we introduce an external stack that contains all the already visited states in the current execution trace. Each state is stored as a struct whose contents are defined during the instantiation process, as shown in Listing 3.2. States are organized into pages that can be allocated dynamically as needed. Figure 3.7 shows an overview of this page-like structure for storing the stack of states.

Variables table

State data is spread across several places: Spin's state vector holds the values of the current state, and the state stack contains all states received at the moment. Some times it is necessary to retrieve or update the values in a generic way. For instance, reports from the system with state data may contain any subset of these variables. To support this, we keep a table with relevant meta-data about the system variables, which is initialized once using the configuration provided by the user.

Figure 3.8 shows the structure and contents of the variables table. In particular, for each variable the table stores:

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

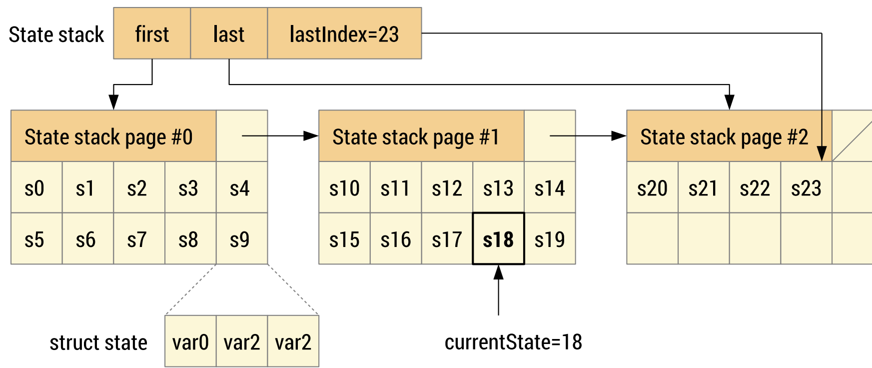


Figure 3.7: State stack

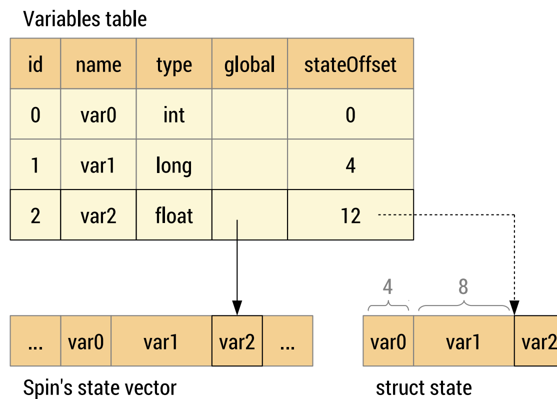


Figure 3.8: Variables table

- **id.** Numeric identifier assigned to the variable. This is the identifier that will be used in the communications with system executions. This includes the following messages: *VariablesDeclaration* (where the correspondence between ids and names is established), *VariableSelection*, *ParameterSetup*, and *Report(Measure)*.

- **name.** String with the name of the variable.

- **type.** An enum value with the type of the variable. This is used to automatically select the appropriate function to operate on each variable.

- **global.** A pointer to the position in Spin's state vector where the current value of the variable is stored.

- **stateOffset.** An integer with the offset (in bytes) of the variable within the struct state.

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

This data is used to handle the variables in a generic fashion where possible. For instance, when receiving a new state from the system, the data must be pushed to the back of the state stack. The `type` and `stateOffset` values tell the analyzer where and how much data must be copied to the new struct state. Later, when the state is copied from the stack to the global state, the `global` pointer indicates where each value should be placed.

3.4.3 Exploration and analysis

This section explains how the exploration and analysis of the state space of the system under test is carried out using the Promela template. Once the data structures have been initialized, the main responsibilities of the analyzer are generating new parameter configurations, and analyzing system execution traces using these configurations. This process can be seen as two nested loops: one generating new configurations and another receiving the states of the execution associated with that configuration.

Listing 3.3 shows the initial phases of the Promela template, which starts at the special `init` process. After the initialization, the `explore` inline contains the bulk of the analyzer. A new parameter configuration is generated in `generateConfig` (explained later). If the configuration passes the checks imposed by the user, a new execution is started and analyzed inside `executeSystem`. Required cleanup follows, whether the system was executed or not. These steps are repeated until the parameter space is exhausted.

Parameter configuration generation

The process of generating a new complete parameter configuration yields the "outer loop" of the exploration and analysis. This loop is not coded explicitly, but rather it takes advantage of Spin's exhaustive exploration of indeterministic choices.

The code for configuration generation is shown in Listing 3.4. For each parameter `param`, the template provides a `generate_param` inline. These inlines include a single `if` with multiple indeterministic choices, one with each of the values specified by the user. Ranges are converted automatically into individual values as well. Then, inline `generateConfig` calls all these `generate_x` inlines in sequence. Spin's depth-first search will pick one of these choices and continue exploring, but it will later return to that point to follow another path, until all of them are exhausted. Thus, the `executeSystem` inline from Listing 3.3 will be called once for each complete parameter configuration.

Trace reconstruction and analysis

Once a new parameter configuration has been generated, the system can be executed under this configuration, and the resulting execution trace can be analyzed. Listing 3.5

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

```
inline explore() {
    generateConfig();
    cleanupPrevExecution();
    storeConfig();

    if
    :: c_expr { checkConfig() } -> executeSystem()
    :: else -> skip
    fi;

    cleanupExecution();
}

inline initialize() {
    c_code {
        os_init();
        os_launchServerSocket();
        initializeDatabase();
        initializeStateStack();
        initializeVariableTable();
    }
}

init {
    initialize();
    explore();
}
```

Listing 3.3: Promela template: initialization and exploration

shows part of this process. We omit the definitions of inlines `setupNewExecution` and `startExecution`, which launch the system execution, perform the initial negotiation and setup phases of the communication protocol (as described in Section 3.2), and reset some analyzer variables.

The most important part of this process is contained in the `analyzeExecution` inline. We have omitted some details from this inline, including checking assertions, and left only the trace reconstruction core. This core is quite simple: a loop that will retrieve the next state and update Spin's global state accordingly, while there are more states available. The `nextState` functions hides the origin of this state: it may be a new state from the system execution trace, or it may be a state that has already been visited because Spin backtracked during the analysis of the current trace. The `currentState` variable, which is stored in the state vector, stores the index of the current state. When the analyzer requests the state that follows the current one, it first checks whether it is contained in the state stack. If not, the stack is updated first with a new state from the execution trace, keeping the values of the variables that did not change. Then, the variables in the state

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

```
1  #foreach( $parameter in $parameterList )
2  inline generate_${parameter.Name}() {
3      if
4      #foreach( $value in $parameter.Values )
5      :: c_code { now.$parameter.Name =
           $ValuesHelper.normalize($value, $parameter.Type); }
6      #end
7      fi
8  }
9  #end
10
11 inline generateConfig() {
12     #foreach( $parameter in $parameterList )
13         generate_${parameter.Name}();
14     #end
15     skip
16 }
```

Listing 3.4: Promela template: configuration generation

vector are updated from the values of the corresponding state in the state stack.

Note that this only reconstructs the execution trace as a series of Spin states. The analysis of temporal properties is performed over these states by the analyzer generated by Spin, without further assistance from the template. However, to associate the proper result with each execution trace, the Promela template must be aware that Spin may truncate the analysis of an execution trace if a temporal property is checked. Thus, the `cleanupPrevExecution` shown on Listing 3.3 is introduced to deal with this situation, using global data stored outside of the state vector (not shown in these listings). This inline will be executed after another complete configuration has been generated, which is the first single point where it can be known whether the last trace was truncated.

3.4.4 Protocol implementation

The communication protocol between the analyzer and the system, described in Section 3.2, has been implemented using Google Protocol Buffers [14]. Protocol buffers are a mechanism for serializing structured data, typically for storage or exchange between heterogeneous parties. This structured data is defined as one or more protocol buffer messages in a `.proto` file. Each message contains a series of name-value pairs, and messages may be nested. Protocol buffer messages can be encoded either as plain text or in a efficient binary format.

Compared to plain XML, protocol buffers are simpler to use, and more efficient in terms of message size and processing time [94]. The ASN.1 standard [8], another popular choice for message exchange, features a definition language which is more expressive

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

```
c_code {
    void nextState(int fd) {
        now.currentState++;
        if (now.currentState > lastState) {
            copyPrevState();
            readState(fd);
        }
        else {
            // Backtracked
        }
        updateSpinStateFromStateStack();
    }
}

inline analyzeExecution() {
    do
    :: (running) -> c_code {
            nextState(os_getProxydata()->clientSocket);
        };
    :: (!running) -> break
    od
}

inline executeSystem() {
    setupNewExecution();
    startExecution();
    analyzeExecution();
}
```

Listing 3.5: Promela template: trace reconstruction and analysis

than protocol buffers, and several encoding formats. However, the simplicity and ease of use of protocol buffers were valued over advanced ASN.1 features.

The official distribution supports Java, C++ and Python, but not C, and thus is not directly embeddable in our Promela template. For this purpose we used protobuf-c [29], a project which provides C bindings for protocol buffers.

Given the original protocol buffer definition file (see Appendix B), protocol buffer compilers generate bindings tailored for handling the messages. Each set of bindings is targeted at a particular programming language, such as Java or C++, and is supported by a library provided with the compiler. We developed additional helper classes and functions on top of these generated bindings. Figure 3.9 shows an overview of this process of generating bindings from the protocol message definitions.

Protocol buffer messages are not auto-delimited, thus requiring some sort of convention to adequately parse messages sent over a stream. In our implementation we always send a *Header* message of fixed length preceding each message. *Header* includes

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

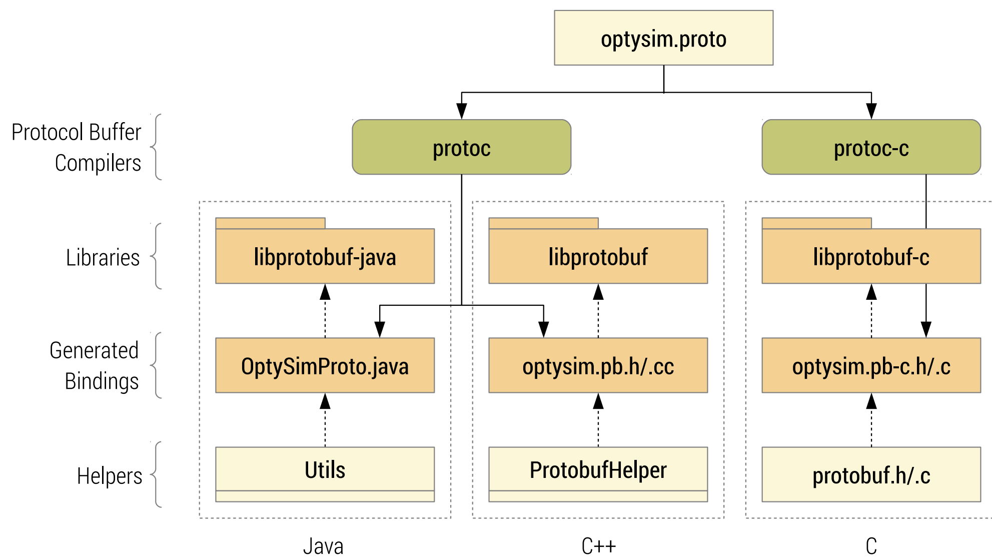


Figure 3.9: Generating bindings from protocol buffer message definitions

meta-information of the following message, including size (in bytes) and type. This simplifies the task of reading new messages whose length is unknown *a priori*.

To handle these headers in a transparent way, we developed additional C functions on top of the ones generated by protobuf-c. These functions can be grouped into three layers:

- **os_sendHeader/os_readHeader.** Send or read a single *Header* message. The required information is passed directly as parameters, lifting the burden of building or parsing the message.

- **os_sendMesssageWithHeader/os_readMesssageWithHeader.** Send or read a message with the corresponding header. In the former, a new *Header* message is sent with the meta-data passed in the parameters. In the latter, a *Header* message is read first, and the meta-data returned as output parameters. The message is also decoded using the decoder function passed as argument.

- **os_sendXXXWithHeader/os_readXXXWithHeader.** Functions for sending and reading specific types of messages. These functions handle meta-data extraction for the headers.

Although we could take advantage of multiple threads to read states from the system execution trace in parallel with the analysis, our implementation is sequential. The protocol allows implementations where state data is read on an "as needed" basis, given a sufficiently large reception buffer.

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

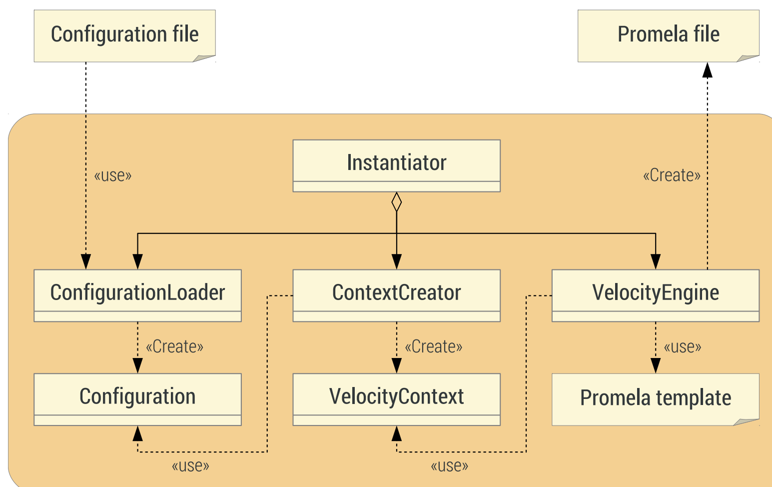


Figure 3.10: Overview of the Promela instantiation process

The negotiation and setup phases are synchronized, where each side knows which message (or messages) to expect and how to respond. During the execution, the analyzer requests a new state after each "inner loop" step of the analyzer. If the following state is not available in the state stack, a new *Report* message is read from the system execution. In the event that the system execution ends or it encounters an error, the analyzer is not required to respond in a timely manner to the *Report* message that announces this. If the analyzer truncates a trace, a *Command(Stop)* message is sent during the cleanup step. Any *Report* messages sent by the system meanwhile are ignored. In this case the analyzer does wait for the system to send a *Report(Stopped)*.

3.4.5 Template instantiation

The template instantiation process was implemented using the Velocity Java library and JAXB [13]. We use the latter to parse configuration files written as XML files (see Appendix A). Using JAXB we also generated a series Java classes from a XML Schema definition that represent the elements of a configuration file. On top of that, we developed a series of extensions for some of the elements, such as properties or ranges or values, that provide additional methods that are particularly useful for the instantiation process.

The instantiation process is shown in Figure 3.10. The `ConfigurationLoader` class loads the file provided by the user and produces a `Configuration` instance with its contents. We implemented several subclasses of `ConfigurationLoader` for parsing different types of configuration files, such as `XmlConfigurationLoader` for loading XML files and `LegacyNs2ConfigurationLoader` for parsing an earlier version of the configuration files which were attached to ns-2 OTcl scenario files.

The `ContextCreator` class is tasked with preparing a `VelocityContext` instance

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

with information from the `Configuration`. `VelocityContext` objects are used in Velocity to pass variables and other information to the template. For instance, the `$allVariablesList` from Listings 3.1 and 3.2 is a Java `List` instance with variable meta-data obtained in the previous step. As we experimented with several versions of the Promela template, we also developed different subclasses for each.

Last, the `VelocityContext` instance and a reference to the Promela template file are passed to a standard `VelocityEngine` instance, which then produces the final Promela file. This Promela file is ready to be processed by Spin and compiled into the executable analyzer.

3.4.6 System integration

The descriptions of the previous sections mostly concern the implementation of the analyzer. However, the systems under analysis also require a front-end that handles the communication with the analyzer and facilitates the extraction and abstraction of execution traces. For our case studies, we implemented these front-ends as libraries that integrate with the system. Some of these front-ends are not completely transparent to the system, and thus require some additional effort in order to be used for a particular scenario, e.g. by indicating manually the variables that will be part of the execution trace.

We developed libraries for use with ns-2 and Java systems. The former were built as the combination of a generic C++ library plus an system specific library for ns-2. This allows easier porting to other C++-based systems, such as ns-3. The generic C++ library, and ns-2 integration library are described in Chapter 5. On the other hand, for Java systems we implemented a front-end using the Java Debugger Interface (JDI), which is transparent to the system. This integration also provides a plug-in for the Eclipse IDE, and is described in Chapter 7.

3.5 Summary

This chapter has presented OptySim, our framework for analyzing systems whose behavior can be observed as an execution trace. This allows the analysis of systems whose state space is too big to be covered exhaustively, or which are presented as black boxes. The analysis and supported objectives are flexible enough to be used for different purposes, such as testing or performance optimization.

As part of this framework, we proposed a protocol for communicating the analyzer and the system execution that can be adopted by other systems. The chapters in Part III include the description of libraries that can be used as a starting point for this adoption. Furthermore, this protocol may be adopted by other analyzers whose inputs can be thought as execution traces.

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

Chapter 4

Abstraction of execution traces

In previous chapters we have talked about system states, execution traces and abstract traces, but we have not discussed how properties are checked on these execution traces or formalized our abstractions yet.

This thesis is focused on the analysis of heterogeneous systems whose observable behavior can be described as a series of states, which we call a trace. Each state is composed of one or more variables, whose values and evolution along a trace can be used to check relevant objectives described as properties, such as LTL formulas. These traces are generated by executing the system under different circumstances, such as setting different values for system parameters. Usually, a large number of different execution traces is generated from the same system by changing the available parameters, or just by the influence of external factors.

Analyzing complete traces for complex systems may require significant resources, and even more when analyzing a large set of traces. However, not all the information contained in a complete trace is relevant to the objectives that a user may want to check. We can reduce the overhead of the analysis and communication by abstracting the trace, i.e. working with a reduced version of the trace which is still useful for the objectives at hand. We developed two such abstractions, called *counter* and *hash projections*, which differ in their computational complexity and on the kinds of properties that can be checked with them.

In addition, we developed an exploration method for state spaces composed of independent execution traces. When analyzing a trace, it is often useful to stop at the first objective (either positive or negative) that was met. In OptySim, independent execution traces are part of the same state space. Asking a model checker to find all property violation will probably result in several violations reported for each trace. We tweak the depth-first algorithm described in Chapter 2 so that exploration continues with the following trace after an objective is met on a trace.

This chapter defines our view of execution traces, and presents our trace abstractions for reducing the cost of analysis. We also show how these abstractions preserve the

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

results of analysis, compared to the corresponding complete traces. Finally, we describe how our modified exploration method for trace-based state spaces works.

4.1 State space

In this section we discuss our definitions of state space and execution traces, relative to a system under analysis.

Let *Sys* be the system under analysis. This may represent different kinds of systems, e.g. a computer program or a model of a real system, which can be described at a high level in a similar way. *Sys* contains an enumerable set of observable features *Var*, which we call variables. We assume that these variables have a known domain, e.g. booleans, natural numbers, real numbers, and that these domains are either finite or discretized. A state is a valuation of the variables of the system.

Definition 4.1 (State) *Let Sys be a system with variables Var, Val_v the finite set of possible values of $v \in Var$, and $Val = \bigcup_v Val_v$ the union of theses sets. A state σ is a function $\sigma : Var \to Val$ that associates each variable with its value. Let us denote with $State$ the finite set of possible states of a system Sys.*

A system may be *executed*, causing its initial state to change during the execution. This evolution of the system state may be discrete, e.g. the execution of an instruction in a program, or discretized into observable steps, e.g. an environmental variable in a real system measured every second. The evolution of the state of the system in discrete steps for a given execution is what we call a *trace*. Even with a finite-state system, a trace may be finite or infinite, e.g. if the trace contains an infinite loop.

Definition 4.2 (Trace) *Let Sys be a system with states $State$. A trace t of Sys is a sequence of states:*

$$t = \sigma_0 \to \sigma_1 \to \sigma_2 \to \ldots \in Trace \tag{4.1}$$

with Trace being the set of all possible infinite sequences of elements from $State$, i.e. the set of all possible traces.

The initial state of the system, σ_0, is partially defined by a subset of *Var* called *Param*, or parameters, whose values can be assigned before the system is executed. Different initial states will usually lead to a different evolution. The evolution of the system during an execution may also be determined by some external factors which cannot be controlled (to a certain degree) in a "real" environment. External factors include random number generators or interferences in a wireless network.

The influence of the parameters and the external factors in the execution of the systems means that the execution of a system may produce a (possibly infinite) number of different execution traces. This set, called *Trace*, constitutes the state space in which we are interested regarding the analysis of a system.

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

Definition 4.3 (System) *A system Sys is a tuple* $\langle Var, Param, State, Extern, Initial, Exec\rangle$ *where:*

- *Var is a finite set of variables of the system, with each variable v belonging to a finite domain of possible values* Val_v
- $Param \subseteq Var$ *is the subset of configurable parameters*
- *State is a set of functions* $\sigma : Var \rightarrow Val \in State$ *that assign values to variables*
- *Extern are the external factors that affect the execution of the system*
- *Initial is the initial configuration of the system, including the initial values of Var*
- $Exec : \sigma, Extern \rightarrow \sigma'$ *is the function that represents the behavior of the system, producing a new state σ' from a previous state σ and the influence of Extern.*

4.1.1 Parameter space

The values assigned to the *Param* subset determines part of the initial state of a system *Sys*, and thus the trace that will be generated as a consequence of its execution. To generate a greater number of different traces, and thus achieve a greater coverage of the state space of possible traces, we may declare a subset of the possible values of the variables in *Param*, and execute the system for each of these values. Depending on the system and on other external factors, different initial configurations may yield different traces.

Definition 4.4 (Initial configuration) *Let* $Param \subseteq Var$ *be the subset of parameters of a system Sys. The initial configuration is a function* $\sigma^0 : Param \rightarrow Val$ *that assigns a value to each parameter. The values given by a initial configuration are part of the initial state of the system:*

$$\forall p \in Param, \sigma_0(p) = \sigma^0(p) \tag{4.2}$$

Constructing the parameter space

A set of initial configurations may be constructed by declaring the sets of possible values for each of the parameters in *Param*, and generating all possible combinations. Different parameters may have different data types with different value domains, some of them discrete and some continuous. We focus on discrete and discretized value domains.

Definition 4.5 (Initial values for a parameter) *For each parameter* $p \in Param$*, we define the set of initial values for p,* $\Sigma^0 : Param \rightarrow 2^{Val}$*, as a subset of all the possible values for the variable:*

$$\forall p \in Param, \Sigma^0(p) \subseteq Val_p \tag{4.3}$$

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

For convenience, the $\Sigma^0(p)$ sets can be declared as a sequence of individual values, as a range of values, or any combination of the two. Ranges have an initial value, a final value, and number of values to be generated in the range, equidistant. This last element is required for continuous domains, but optional for discrete ones. In the latter, if it is omitted, all possible values between the initial and final value will be included in the range.

Definition 4.6 (Initial configuration from parameters initial values) *Let $Param \subseteq Var$ be the subset of parameters of a system Sys. Let Σ^0 be the function that assigns a set of initial values for each of the parameters $p \in Param$. We define InitConfig as the set of all initial configurations σ^0 which is constructed with the product of the individual $\Sigma^0(p)$ sets:*

$$InitConfig = \prod_{p \in Param} \Sigma^0(p) \tag{4.4}$$

In addition, certain combination of parameter values may be prevented from being included as part of σ^0. Depending on the system or on analysis requirements, certain initial configurations are not required to considered or do not make sense. These restrictions may be expressed as boolean conditions over the values of the parameters.

4.1.2 External factors

The behavior of a system may be influenced by external factors over which little or no control can be exerted. In some circumstances, some of these external factors may be emulated in a controlled environment, e.g. a mobile phone connected to an emulated network. External factors may affect the generation of the state space in an uncontrollable manner.

One of the most significant issues in model checking is the so-called state space explosion problem. Many closed systems are already complex enough to yield a state space that is too big to cover in practice, and the introduction of unforeseeable factors is bound to make matters only worse. Thus, a compromise is usually made in order to analyze a significant portion of the state space within reasonable resource limits. For instance, for many uncontrollable external factors this could mean executing the system with the same parameters more than once in order to capture different external effects in each execution. Other external factors, such as thread scheduling, can be influenced to behave in a different way.

Typical external factors include random number generators, task scheduling and environmental factors. In the rest of this section we will review some of these, including their effect in the system execution, and how they can be measured or influenced.

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

Random number generators

Certain systems may include random behaviors, either intrinsically or to simulate unpredictable conditions. These behaviors are usually implemented using random number generators.

Random number generators can be divided into two methods. "True" random numbers depend on some measurable physical phenomenon which is expected to be naturally random, such as atmospheric or thermal noise. On the other hands, pseudo-random number generators can be obtained using special computational algorithms, which depend on an initial value called seed. Given the same seed, the sequence of numbers generated by such algorithms would be exactly the same.

In systems with a pseudo-random number, it may be useful to execute the system with different seeds to try to generate different execution traces. For some types of analysis, it may also be useful to include the seed in *Param*, so that the seed is associated with each execution, allowing the repetition of any system execution.

Task scheduling

Some systems may be composed of several concurrent tasks, such as threads in a computer program or networked applications in a network simulation. These tasks have to be scheduled on one or several processing units, e.g. processor cores or simulator executors, in order to be executed. The scheduling policy decides how these tasks are given time slots for their execution, and may be deterministic or have a certain degree of randomness. In some systems, these tasks will compete with other tasks that co-exist in the same environment, while in others the scheduling of the system tasks is isolated.

Scheduling affects how the traces are generated, since the task that has been scheduled for a given period time is the one generating new states during that time. Also, tasks usually share some common data, and the order in which this data is accessed influences future evolutions of the system.

In our work we assume interleaving semantics for concurrent tasks. That is, several tasks may be scheduled concurrently, but only one is executed at any given time. The effective result is an interleaving of the instructions of each concurrent task. Even with this interleaving model, synchronization between tasks and shared data is a complex concern.

The extension of the state space that can be generated by a system may be greatly expanded by the effects of scheduling. Exploring every possible scheduling of a system can be an unfeasible task for most real cases. Techniques such as partial order reduction [100][67][46] reduce the number of different schedulings that have to be considered from a certain point of view. In the case of partial order reduction, the instructions of a system that do not access shared data can be considered as if executed atomically, and the scheduler does not interrupt them to schedule another task.

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

There are several ways in which task scheduling can be influenced. Some operating systems expose scheduling parameters which can be tweaked in a programmatic manner. The influence of other tasks that are being executed concurrently in the same environment can be limited by running the system on a separate environment. Finally, some systems may be instrumented to introduce thread scheduling system calls, such as `yield` or `sleep`, in key locations, with additional parameters for controlling them, which will force different schedulings of the same tasks.

Environmental factors

Systems that are executed in a real environment may be subject to a whole range of additional factors that impact their behavior in ways that may not be easily predicted. For instance, an application in a mobile device can be affected by other applications in the same device, battery levels, interferences in the wireless link with the base station, occupation of the wireless link by other devices, etc.

The simplest way of partially capturing the influence of these environmental factors is executing the system several times using the same initial parameter configuration. Some of these environments may be partially emulated, e.g. with testing equipment. This may require additional integration with the analysis framework in order to control these environments in a useful way.

4.2 Trace projection

A trace t, as defined in Section 4.1, represents a possible execution of a given system *Sys*. However, we do not usually require (or want) the full trace to be analyzed by Spin. A *projection* of the trace is enough in most cases. For instance, when checking an LTL formula, the part of a trace that is involved in its evaluation may be the only part needed. A projection includes only part of the states of the full trace, and the contents of each state may be partially reproduced as well. In any case, the generation of the full execution trace itself is not affected by the projection.

We now describe how the projection of states is constructed and the correctness relation between the evaluation results regarding the original traces, and the projected ones on Spin. In what follows, we use the same LTL semantics as Spin, without the *next* operator as usual. Note that in $t \models f$, t may be a prefix of a complete trace, i.e. it may not be necessary to generate the whole trace in order to check the satisfaction of a property.

Definition 4.7 (Projection) *Given a subset of variables $V \subseteq Var$, we define the projection of a state σ onto V as the function $\rho_V(\sigma) : V \to Val$ such that $\forall v \in V.\rho_V(\sigma)(v) = \sigma(v)$. Now, given a trace $t = \sigma_0 \to \sigma_1 \to \sigma_2 \to \ldots$, we define the projection of t onto $V \subseteq Var$ as*

$$\rho_V(t) = \rho_V(\sigma_0) \to \rho_V(\sigma_1) \to \rho_V(\sigma_2) \to \ldots \tag{4.5}$$

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

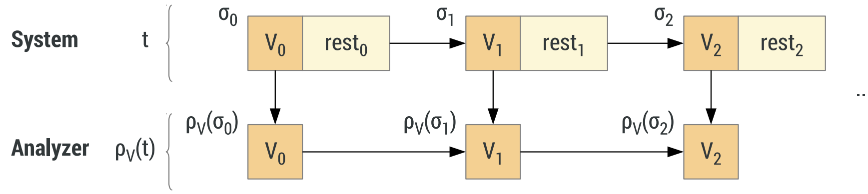


Figure 4.1: Trace projection

Figure 4.1 shows the projection ρ_V of a trace. Observe that V divides each state σ_i into two parts: the part concerning the variables of V in state i (V_i), and the rest (rest_i). The projection simply takes the former part from each state and ignores the latter.

The effect of this projection is similar to that of the *cone of influence* technique [45] or Spin's *selective data hiding* [80]. However, while these techniques simplify the model of the system to include only variables which are on the set V (or which influence them) before executing it, we execute the system as is and then simplify (i.e. project) the generated trace. However, we do not automatically include other variables not in V.

As a general result of this definition of projection, if all the variables required for evaluating an LTL formula are present in the projection, the evaluation of the formula is not affected. Let f be an LTL formula and let us denote the set of variables in f as $vars(f)$.

Proposition 4.1 *Given a trace t, a temporal formula f and a subset of system variables $V \subseteq Var$, if $vars(f) \subseteq V$ then*

$$t \models f \iff \rho_V(t) \models f. \tag{4.6}$$

In contrast to model checking, we do not check formulas on all the possible traces that can be generated from a system, but rather on a subset of those traces. When using LTL formulas as objectives, it may desirable to check if a formula is checked in all, some or none of this subset of traces. Thus we extend $\models$ for sets of traces and the $\forall$ and $\exists$ quantifier operators.

Definition 4.8 *Given a temporal formula f, a set of traces T, and a trace projection function ρ,*

$$\begin{aligned} T \models \forall f &\iff \forall t \in T.t \models f \\ T \models \nexists f &\iff \nexists t \in T.t \models f \\ T \models \exists f &\iff \exists t \in T.t \models f \end{aligned} \tag{4.7}$$

By Proposition 4.1, the previous definition can also be applied to set of traces when using the ρ_V projection.

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

Due to the elimination of most system variables in the projected states, it is very likely that a projected trace $\rho_V(t)$ contains many consecutive repeated states. This represents a problem for the model checker since it can erroneously deduce that the original trace has a cycle due to the double depth search (DDS) algorithm used by Spin to check properties. Note that this does not contradict Proposition 4.1, since in this result we do not assume any particular algorithm to evaluate the property on the projected trace. In the following sections, we use relation $\models_s$ to distinguish between the LTL evaluation carried out by Spin through the DDS algorithm, and the satisfaction relation $\models$ defined above.

To correctly eliminate consecutive repeated states in traces, we propose two different techniques that we discuss in the following subsections.

4.2.1 Counter projection

A simple solution is to add a new counter variable *count* to the set of visible variables V. This counter is increased for every new state, thus removing the possibility that Spin erroneously finds a non-existing cycle. Observe that this also precludes Spin from detecting real cycles present in the system. This case will be discussed in the following subsection.

We extend the notion of trace projection given in Definition 4.7 by adding the state counter variable as follows:

Definition 4.9 (Counter projection) *Given a subset of visible variables $V \subseteq Var$ and a fresh variable $count \notin Var$, we define the i-th counter projection of a state $\sigma : V \to Val$ as $\rho_V^i(\sigma) : V \cup \{count\} \to Val$ as*

$$\rho_V^i(\sigma)(v) = \begin{cases} \rho_V(\sigma)(v) & \textit{if } v \in V \\ i & \textit{if } v = count \end{cases}$$

Variable count is called state counter of $\rho_V^i(\sigma)$.

Now, given a trace $t = \sigma_0 \to \sigma_1 \to \sigma_2 \cdots$ we define the counter projection of t onto V, ρ_V^c, by projecting each state σ_i with the i-th counter projection, that is,

$$\rho_V^c(t) = \rho_V^0(\sigma_0) \to \rho_V^1(\sigma_1) \to \rho_V^2(\sigma_2) \cdots$$

It is worth noting that there is a slight difference in the notation of the counter projection of a trace (ρ_V^c) and the counter projection of a state (ρ_V^i). Notice that the former includes a c superscript, while the latter includes the value of the counter itself as a superscript. Figure 4.2 shows the projection of a trace with the addition of the state counter.

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

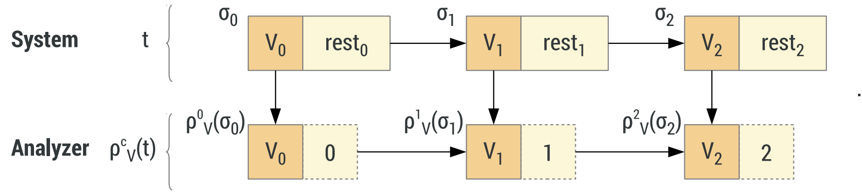


Figure 4.2: Trace projection with state counting

4.2.2 Hash projection

In this section, we assume that system states have a canonical representation, which makes it possible to safely check whether two states are equal. This is a problem that may not be trivial depending on the system, e.g. for languages that make an intensive use of dynamic memory [62]. However, in this section, the actual representation is not relevant for the results obtained. We only need to assume that given two logically equal system states σ_i and σ_j, there exists a *matching algorithm* able to check that they are equal.

We use a proper hash function $h : State \rightarrow Int$ to represent each state in the projected trace. It is worth noting that since not all of the system states σ have to be stored (we only project the visible part $\rho_V(\sigma)$), we may assume that function h is very precise, producing a minimum number of collisions. That is, $h(\sigma_i) = h(\sigma_j) \Longrightarrow \sigma_i = \sigma_j$, with a high degree of probability.

Collision probability of hash function. In our implementation of the hash projection we used the MD5 hash function [6]. Here we present a brief study of the collision probability of this function in a practical case. This function transforms the given input (in this case a string representation of the system state) into a 128-bit digest. Thus, there are 2^{128} (or about 3.3×10^{38}) possible values of this function. We are interested in the likelihood of a *birthday attack* [109], i.e. the probability of a collision between any two states belonging to the same trace or, conversely, in the number of different states that could be generated before a collision is found with a given probability. For instance, if we assume that a collision probability of 10^{-12} is enough for our analysis, this number is approximately 2.6×10^{13}. Given a state size of 64 bytes (a reasonable assumption, e.g. given the experiments in Chapter 7), about 1.5×10^{6} gigabytes of memory would be required to store this number of states. This is well beyond what current computers store, and therefore computationally unpractical. However, further refinements can be applied to the hashing function if necessary, as studied for Spin in [118] and [82].

Now, we extend the notion of state projection given in Definition 4.7 by adding the codification of the whole state (including the non-visible part) as follows:

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

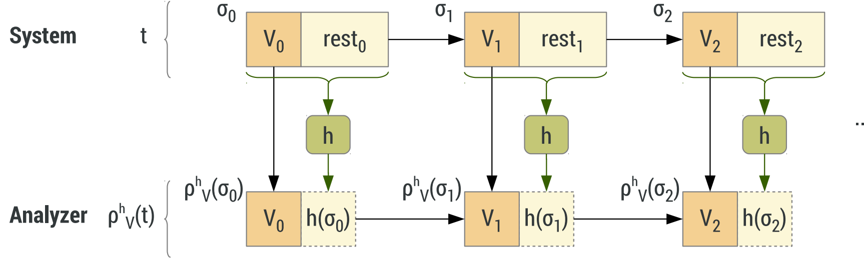


Figure 4.3: Trace projection with state hashing

Definition 4.10 (Hash projection) *Given a subset of visible variables $V \subseteq Var$, and a fresh variable hash $\notin Var$, we define the hash projection $\rho_V^h(\sigma)$ of a state σ onto V using the hash function h as $\rho_V^h(\sigma) : V \cup \{hash\} \to Val$ as*

$$\rho_V^h(\sigma)(v) = \begin{cases} \rho_V(\sigma)(v) & \text{if } v \in V \\ h(\sigma) & \text{if } v = hash \end{cases}$$

Now, given a trace $t = \sigma_0 \to \sigma_1 \to \sigma_2 \cdots$ we define the hash projection of t *onto V, ρ_V^h, by projecting each state σ_i with the hash projection, that is,*

$$\rho_V^h(t) = \rho_V^h(\sigma_0) \to \rho_V^h(\sigma_1) \to \rho_V^h(\sigma_2) \cdots$$

Figure 4.3 shows the projection of a trace with the addition of the state hash. Only projected states $\rho_V^h(\sigma)$ are transferred to Spin. If the model checker detects that two states $\rho_V^h(\sigma_i)$ and $\rho_V^h(\sigma_j)$ are equal, then we can infer that the original states σ_i and σ_j are equal with a high degree of probability.

4.2.3 Folded projections

We propose an optimization to the previous projections to reduce the number of states of the projected trace. To define it, we modify the transition relation $\to$ defined in 4.2 by labeling transitions as follows.

Definition 4.11 (Labeled trace) *Let Sys be a system with states State. A labeled trace t of Sys is a sequence of states:*

$$t = \sigma_0 \xrightarrow{M_1} \sigma_1 \xrightarrow{M_2} \sigma_2 \xrightarrow{M_3} \sigma_3 \ldots \in LTrace \tag{4.8}$$

where each label $M_i \subseteq Var$ is the set of variables which were modified in the transition between states $\sigma_i - 1$ and σ_i, and LTrace being the set of all possible infinite labeled traces.

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

Recall that both counter and hash projections of traces t ($\rho_V^c(t)$ and $\rho_V^h(t)$) discard all system variables except the ones in V, which are the only ones needed to check a temporal formula f, ($V = vars(f)$), while the rest of the state is collapsed into a single variable. However, Spin does not need to know about states in which none of the variables in V change as they do not affect the evaluation of temporal formulas (discussed later in Section 4.3). Thus, we propose removing these states from the final projection given to Spin. We call these removed states *folded states*, and the process of removing them *folding*.

Definition 4.12 (Folded projections) *Given a labeled trace* $t = \sigma_0 \xrightarrow{M_1} \sigma_1 \xrightarrow{M_2} \sigma_2 \cdots$, *we define the folded counter projection of* t *onto* $V \subseteq Var$ *as*

$$\phi_V^c(t) = \rho_V^0(\sigma_{i_0}) \to \rho_V^1(\sigma_{i_1}) \to \rho_V^2(\sigma_{i_2}) \to \dots$$

and the folded hash projection of t *onto* $V \subseteq Var$ *as*

$$\phi_V^h(t) = \rho_V^h(\sigma_{i_0}) \to \rho_V^h(\sigma_{i_1}) \to \rho_V^h(\sigma_{i_2}) \to \dots$$

such that:

1. $i_0 = 0$,
2. *for all* $k \geq 0$, $i_k < i_{k+1}$
3. *for all* $k \geq 1$, $M_{i_k} \cap V \neq \emptyset$.
4. *if there exists* $j > 0$ *such that* $\forall k \geq 0, i_k \neq j$, *then* $M_j \cap V = \emptyset$

That is, the folded projection only contains those states where some visible variable has just been modified.

However, this definition of folding is not enough to allow a precise cycle detection, which was the main reason for introducing the hash projection. If an infinite cycle is located in the folded states, Spin will not be informed of any new system state, and thus Spin will not be aware that the system is going to loop endlessly in those states.

To avoid this, we define the *limited* folding of a hash projection. In this context, limited means that the number of folded states between two non-folded states is never greater than a given limit. After a specified number of folded system states, we project the next system state, even if that state did not change any of the variables in V.

An implementation may choose to use a timer as a limit instead of a state counter, which may be more practical and would not affect the results given below. This projection may be further refined in the implementation with an adaptive limit, e.g. a limit which decreases progressively.

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

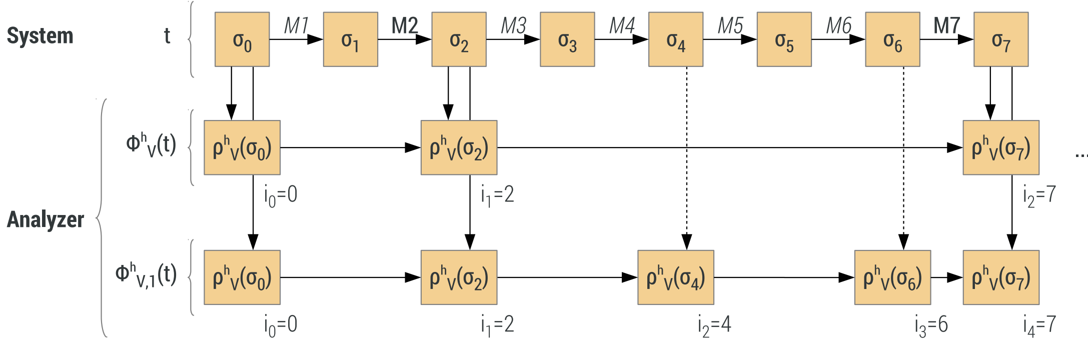


Figure 4.4: Example of the folded and limited folded hash projections of a trace

Definition 4.13 (Limited folded hash projection) *Given a labeled trace* $t = \sigma_0 \xrightarrow{M_1} \sigma_1 \xrightarrow{M_2} \sigma_2 \cdots$ *and a limit* $l > 0$, *we define the limited folded hash projection of* t *onto* $V \subseteq Var$ *as*

$$\phi^h_{V,l}(t) = \rho^h_V(\sigma_{j_0}) \rightarrow \rho^h_V(\sigma_{j_1}) \rightarrow \rho^h_V(\sigma_{j_2}) \rightarrow \ldots$$

such that:

1. $j_0 = 0$,
2. *for all* $k \geq 0$, $j_k < j_{k+1}$, *and*
3. *for all* $k \geq 1$, *either* $M_{j_k} \cap V \neq \emptyset$, *or, the distance between* j_k *and* j_{k-1} *is limit* l, *that is* $j_k - j_{k-1} = l$,
4. *if there exists* $j > 0$ *such that* $\forall k \geq 0, j_k \neq j$, *then* $M_j \cap V = \emptyset$.

Although we could define a limited folded counter projection in a similar fashion, there would be no benefit in doing so since the counter prevents any kind of cycle from being detected.

Figure 4.4 shows an example of a limited folded hash projection, with limit $l = 1$. This limit ensures that only one state can be folded consecutively. In this figure, a bold M_i label indicates that $M_i \cap V \neq \emptyset$, i.e. that transition modifies one or more variables in V. In this example, the limit forces the projection of states σ_4 and σ_6, which should have been folded, resulting in the projected states σ_{i_2} and σ_{i_3}.

4.3 Preservation of results

We now discuss how the results are preserved regarding the satisfaction of temporal properties in the original traces and in the projected traces. Here we assume that the

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

algorithm for checking the satisfaction of a property uses the double depth search algorithm as implemented by Spin (see Section 2.1.5).

First, we focus on the preservation of results using the counter and hash projections as described in Definitions 4.9 and 4.10, respectively, which were introduced to deal with cycles as required by the model checking algorithm implemented by Spin.

Proposition 4.2 *Given a temporal formula f using only the eventually "$\Diamond$" and until "U" temporal operators, if $vars(f) \subseteq V \subseteq Var$ then*

$$t \models f \iff \rho_V^c(t) \models_s f.$$

Counter projection ρ_V^c does not permit Spin to detect cycles in the projected trace. Thus, properties that do not require the detection of cycles (i.e. those that use only operators *eventually* "$\Diamond$" and *until* "U") can be properly checked over this projection. In contrast, since properties that use the *always* "$\Box$" temporal operator are checked by Spin by searching for cycles, they cannot be analyzed over $\rho_V^c(t)$.

Proposition 4.3 *Given a temporal formula f and a set of variables V, if $vars(f) \subseteq V \subseteq Var$ then*

$$\rho_V^h(t) \models_s f \Longrightarrow t \models f$$

with the degree of probability allowed by h, and

$$t \models f \Longrightarrow \rho_V^h(t) \models_s f.$$

This theorem asserts that any temporal formula which is satisfied in the original trace t, is also satisfied in the hash projection of the trace. The converse, while generally true for practical purposes, is limited by the quality of the hash function h. In addition to projecting the variables in f, as established in Proposition 4.1, the hash projection includes a variable computed by h that identifies the global state and is used to detect cycles in the trace.

Now, we show how the results are preserved with the folded projections described in Section 4.2.3.

Proposition 4.4 *Given a temporal formula f using the eventually "$\Diamond$" and until "U" temporal operators, and a set of variables V, if $vars(f) \subseteq V \subseteq Var$ then*

$$t \models f \quad \iff \quad \phi_V^c(t) \models_s f.$$

This result is not affected by the folding in the projection, because i) the folded states are not required to evaluate the boolean tests of the temporal formula, and ii) cycle detection is not affected since it is not supported by the counter projection, as discussed in Section 4.3.

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

Proposition 4.5 *Given a temporal formula f and a set of variables V, if $vars(f) \subseteq V \subseteq Var$ then*

$$\phi^h_{V,l}(t) \models_s f \quad \Longrightarrow \quad t \models f$$

with the degree of probability allowed by h, and

$$t \models f \quad \Longrightarrow \quad \phi^h_{V,l}(t) \models_s f.$$

Again, this result is not affected by the folding in the projection thanks to the limit, which covers the detection of cycles in (otherwise) folded states. If there is a cycle in an infinite sequence of states the transition labels of which are $M_i \cap V = \emptyset$, the limited folding only removes a subset of the states. Since a cycle is by definition a finite sequence of states, it is guaranteed that eventually two equal states will be projected, and thus the cycle will be detected.

4.4 Operational semantics of executions exploration

In our trace-based approach we analyze execution traces, which are sequences of states. Although an execution trace is lineal, when combined with a never claim process it may result in a set of possible traces, all starting with the same prefix. If Spin finds a property violation in this state space, it will also report the counterexample that led to the error, i.e. a trace that combines both the execution trace from the system and never claim transitions. If Spin is instructed to find more than one error, several different counterexamples may be reported. However, it is often enough to report just the first property violation as the general result for the trace.

Our state space to be analyzed in built by aggregating several independent execution traces in a tree-like structure. The first nodes on the tree perform some setup tasks, including the selection of system parameters. From that point on, independent execution traces hang from different “root” nodes. If Spin is instructed to stop on the first error, many execution traces can be left without exploring. If all errors are to be reported, however, many property violations may be found within each trace. If we are only interested in the first error of each trace, this approach does not work.

We propose that in these cases, where the state space is an aggregation of independent execution traces with a common “setup” prefix, only the first property violation in each trace is reported. After one is found in an execution trace, the analysis of the current trace should stop and continue with the next. Given that execution traces are generated and analyzed on-the-fly, stopping the analysis of one early can yield significant resource savings, since the system itself can also be stopped earlier.

We extend the operation semantics for Promela described in [64] to formalize this exploration mode. The semantics in [64] is geared towards abstract model checking, including parameters τ and φ that define the evaluation of boolean expressions and effect

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

of instructions, respectively. In this section, we assume that values of these parameters are τ^P and φ^P, i.e. the meaning implemented by Spin.

Let M be a Promela model, and $\Gamma = \{\gamma \mid \gamma : Pid \rightarrow State \cup \{\perp\}\}$ the set of model configurations. Let Pid be the set of process identifiers, with $n \in Pid$ the identifier of the never claim process. A configuration $\gamma \in \Gamma$ associates each process identifier $j \in Pid$ with its state $\gamma(j) \in State$. If $j \in Pid$ does not represent a process instance in γ, then $\gamma(j) = \perp$.

Our trace exploration semantics requires the depth-first search algorithm (see Section 2.1.3) to be explicit. That is, we must enforce that the exploration continues with the portion of the state space that corresponds to the next system execution trace when a property violation is found in the current one. To this end, instead of working with traces composed of states as in [64], each step in an exploration trace is defined as a 3-tuple containing the stack of states from the root and up to the current state, a list of continuations (unexplored children for each state in the previous stack), and a set of visited states. Let $State^*$ be the set of ordered sequences of states, and $State^{**}$ the set of ordered sequences of $State^*$. Let $Step = State^* \times State^{**} \times 2^{State}$ be the set of all possible steps, and $\langle[\gamma_0,\ldots,\gamma_i],[c_0,\ldots,c_i],v\rangle \in Step$ a step in a exploration trace. If γ_j is the j-th state of a step, then c_j is the continuation list associated with that state. v is the set of visited states.

Note that the initial contents of the second component of an exploration step, the list of continuations, is a single list with the children of the root state. When a new state is explored, its children are appended to this list in a new list. The order in which these continuation lists are generated determines the exploration of the state space, i.e. the same ordering should yield the same results.

In the following, we will represent lists using square brackets, e.g. $[\gamma_0,\ldots,\gamma_{i-1},\gamma_i]$. The empty list will be represented as $[]$, while $[\gamma|l]$ represents a list whose head is γ, and the rest of elements is the (possibly empty) list l. Sets will be represented using curly braces, e.g. $\{a,b\}$, and the empty set as $\{\}$.

To declare the state space that will be explored with our rules, we take advantage of the verification-level rules defined in [64] and reproduced on Figure 4.5. These rules define the transition relation $\longmapsto_{ver}$, which in turn is defined in terms of the transition relations $\longmapsto_{sim}$ and $\longmapsto_{nev}$, which model the execution of the system and the never claim process, respectively. Note that given a system configuration $\gamma \in \Gamma$, $\gamma[\sigma/j]$ denotes the configuration that is equal to γ for all processes, except for process j, whose state has been updated to σ. Also note that, given a configuration γ, $\gamma[\perp/n]$ represents the system part without the never claim process.

Rule `Discard-ver` is triggered when the never claim cannot progress, which means that the current trace cannot be identified by the never claim process and should be discarded. On the other hand, `End-ver` deals with the termination of the system processes. Rule `Synch-ver` defines the synchronized execution of the system processes and the

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

$$\texttt{Discard-ver}\quad \frac{\gamma(n) \neq \perp,\ \gamma \not\longmapsto_{nev}}{\gamma \overset{discard}{\longmapsto}_{ver} stop}$$

$$\texttt{End-ver}\quad \frac{\gamma \overset{end}{\not\longmapsto}_{nev},\ \gamma[\perp/n] \overset{end}{\longmapsto}_{sim} stop}{\gamma \overset{end}{\longmapsto}_{ver} stop}$$

$$\texttt{Synch-ver}\quad \frac{\gamma[\perp/n] \overset{inst}{\longmapsto}_{sim} stop,\ inst \in ProcError \cup \{ies\}}{\gamma \overset{synch_j}{\longmapsto}_{ver} \gamma''[\gamma'(n).\sigma_e[j/_last]/n]}$$

$$\texttt{Claim-ver}\quad \frac{\gamma \overset{end}{\longmapsto}_{nev} stop}{\gamma \overset{claim_v}{\longmapsto}_{ver} stop}$$

Figure 4.5: Selected verification-level rules from [64].

never claim process. Finally, `Claim-ver` detects and labels never claim violations. Missing from these rules is the detection of accepting cycles, which in [64] is defined over the generated trace.

We now define some functions that will be used in our rules. The function $next(\gamma)$, used to explore the state space, is defined as follows:

$$next(\gamma) = \begin{cases} claim_v & \text{if } \gamma \overset{claim_v}{\longmapsto}_{ver} \\ [\,] & \text{if } \gamma \overset{end}{\longmapsto}_{ver} \text{ or } \gamma \overset{discard}{\longmapsto}_{ver} \\ [\gamma' | \gamma \overset{synch_j}{\longmapsto}_{ver} \gamma'] & \text{else} \end{cases} \tag{4.9}$$

This function returns either a list of states that can be reached from γ (in some order), or the label *claim_v* if a claim violation is found by the never claim process ($\gamma \overset{claim_v}{\longmapsto}_{ver}$). Note that this function can return an empty list when the system reaches its final state ($\gamma \overset{end}{\longmapsto}_{ver}$) or the never claim process does not recognize the current trace ($\gamma \overset{discard}{\longmapsto}_{ver}$).

Let $accept(\gamma)$ be a function that returns whether configuration γ is labeled as an accepting state. We overload this function for lists of states, to check whether the given list is an accepting cycle, i.e. it starts and ends with the same state, and it contains and accepting state:

$$accept([r_0,\ldots,r_i]) = (r_0 = r_i) \wedge \exists 0 \leq j \leq i.accept(r_j). \tag{4.10}$$

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

$$\texttt{Expl-ver'} \quad \frac{c_i = [\gamma | c_i'],\ \gamma \notin v,\ c = next(\gamma),\ c \neq claim_v}{\langle [\gamma_0, \ldots, \gamma_i], [c_0, \ldots, c_i], v \rangle \xrightarrow{expl}_{ver'} \langle [\gamma_0, \ldots, \gamma_i, \gamma], [c_0, \ldots, c_i', c], v \cup \{\gamma\} \rangle}$$

$$\texttt{Back1-ver'} \quad \frac{c_i = [\gamma | c_i'],\ \gamma \in v,\ \gamma \notin [\gamma_0, \ldots, \gamma_i]}{\langle [\gamma_0, \ldots, \gamma_i], [c_0, \ldots, c_i], v \rangle \xrightarrow{back}_{ver'} \langle [\gamma_0, \ldots, \gamma_i], [c_0, \ldots, c_i'], v \rangle}$$

$$\texttt{Back2-ver'} \quad \frac{\begin{array}{c} c_i = [\gamma | c_i'],\ \gamma \in v,\ \gamma \in [\gamma_0, \ldots, \gamma_i], \\ \nexists 0 \leq j \leq i.(\gamma_j = \gamma \wedge accept([\gamma_j, \ldots, \gamma_i, \gamma_j])) \end{array}}{\langle [\gamma_0, \ldots, \gamma_i], [c_0, \ldots, c_i], v \rangle \xrightarrow{back}_{ver'} \langle [\gamma_0, \ldots, \gamma_i], [c_0, \ldots, c_i'], v \rangle}$$

$$\texttt{Back3-ver'} \quad \frac{c_i = [\,],\ \neg setup(\gamma_{i-1}) \vee (setup(\gamma_{i-1}) \wedge setup(\gamma_i))}{\langle [\gamma_0, \ldots, \gamma_{i-1}, \gamma_i], [c_0, \ldots, c_{i-1}, c_i], v \rangle \xrightarrow{back}_{ver'} \langle [\gamma_0, \ldots, \gamma_{i-1}], [c_0, \ldots, c_{i-1}], v \rangle}$$

$$\texttt{Trace-end-ver'} \quad \frac{c_i = [\,],\ setup(\gamma_{i-1}),\ \neg setup(\gamma_i)}{\langle [\gamma_0, \ldots, \gamma_{i-1}, \gamma_i], [c_0, \ldots, c_{i-1}, c_i], v \rangle \xrightarrow{trace_end^{[\gamma_0, \ldots, \gamma_i]}}_{ver'} \langle [\gamma_0, \ldots, \gamma_{i-1}], [c_0, \ldots, c_{i-1}], v \rangle}$$

Figure 4.6: Rules for depth-first exploration of trace-based state spaces (I).

Let $setup(\gamma)$ be a function that returns whether γ is part of the setup prefix of the state space. We overload this function for lists of states, to return the prefix of the given list which belongs to the setup prefix:

$$setup([\gamma_0, \ldots, \gamma_i]) = [\gamma_0, \ldots, \gamma_p].\forall 0 \leq j \leq p.setup(\gamma_j) \wedge \neg setup(\gamma_{p+1}) \tag{4.11}$$

Now we define the rules for our transition relation $\longmapsto_{ver'}$, which describe how our depth-first trace-based exploration is performed. These rules are shown in Figures 4.6 and 4.7. Note that some of these rules add a label to the corresponding transition, with the purpose of tracking the results of the exploration.

Informally, the purpose of these rules is as follows:

- **`Expl-ver'`.** Continue the exploration with a successor of the last state.

Publicaciones y Divulgación Científica
uma UNIVERSIDAD DE MÁLAGA

$$\texttt{Claim-ver'}\quad \frac{c_i = [\gamma|c_i'],\ \gamma \notin v,\ next(\gamma) = claim_v,\quad [\gamma_0,\dots,\gamma_p] = setup([\gamma_0,\dots,\gamma_{i-1},\gamma_i])}{\langle[\gamma_0,\dots,\gamma_{i-1},\gamma_i],[c_0,\dots,c_{i-1},c_i],v\rangle \overset{claim_v^{[\gamma_0,\dots,\gamma_i,\gamma]}}{\longmapsto}_{ver'} \langle[\gamma_0,\dots,\gamma_p],[c_0,\dots,c_p],v \cup \{\gamma\}\rangle}$$

$$\texttt{Accept-ver'}\quad \frac{c_i = [\gamma|c_i'],\ \gamma \in v,\ \gamma \in [\gamma_0,\dots,\gamma_i],\quad \exists 0 \le j \le i.\gamma_j = \gamma \wedge accept([\gamma_j,\dots,\gamma_i,\gamma_j]),\quad [\gamma_0,\dots,\gamma_p] = setup([\gamma_0,\dots,\gamma_{i-1},\gamma_i])}{\langle[\gamma_0,\dots,\gamma_i],[c_0,\dots,c_i],v\rangle \overset{accept^{[\gamma_0,\dots,\gamma_i,\gamma]}}{\longmapsto}_{ver'} \langle[\gamma_0,\dots,\gamma_p],[c_0,\dots,c_p],v\rangle}$$

$$\texttt{End-ver'}\quad \langle[\gamma_0],[[]],v\rangle \overset{end}{\longmapsto}_{ver'} stop$$

Figure 4.7: Rules for depth-first exploration of trace-based state spaces (II).

- **`Back*-ver'`.** Several rules control backtracking under different circumstances. Backtracking is modeled one step at a time, checking the contents of the continuations associated with the current state. When the next state to be explored has already been visited, it is simply removed (rules `Back1-ver'` and `Back2-ver'`), and when the current continuation is empty, the next step moves back to the previous state (`Back3-ver'`). The first two rules deal with different cases. `Back1-ver'` is applied if the next state has been visited, but is not on the current stack. On the other hand, if the state has been visited before on the current stack, but it is not part of an accepting cycle, rule `Back2-ver'` is applied.

- **`Trace-end-ver'`.** A trace has been explored completely, and no property violation has been found. The prefix of the trace (up until the first state that belongs exclusively to the trace) is stored as part of the transition label.

- **`Claim-ver'`.** The never claim has reached its end state, thus recognizing a erroneous trace. The complete trace is stored as part of the transition label, and the exploration backtracks to the first state that belongs to the setup prefix.

- **`Accept-ver'`.** The next state in the current continuation has been visited before in the current stack, and exploring it again would lead to an accepting cycle. The complete trace is stored as part of the transition label, and the exploration

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

backtracks to the first state that belongs to the setup prefix.

- **`End-ver'.`** The current step is at the first state, and there are not further continuations available. The exploration stops here.

As mentioned above, the transitions that identify the result of a single execution trace have an additional label indicating which trace produced the result. Once a complete exploration trace has been generated, the individual trace results can be extracted as follows:

- $\overset{\textit{trace-end}}{\longmapsto}{}^{[\gamma_0,\ldots,\gamma_i]}_{ver'}$: the trace whose *prefix* is $[\gamma_0,\ldots,\gamma_i]$ was completely explored without finding any property violation.

- $\overset{\textit{accept}}{\longmapsto}{}^{[\gamma_0,\ldots,\gamma_i]}_{ver'}$: the trace $[\gamma_0,\ldots,\gamma_i]$ was found to contain an acceptance cycle. Note that the whole infinite trace was not generated, but rather a suffix of the trace that is enough to detect the cycle.

- $\overset{\textit{claim_v}}{\longmapsto}{}^{[\gamma_0,\ldots,\gamma_i]}_{ver'}$: the trace $[\gamma_0,\ldots,\gamma_i]$ was found to end with a never claim violation.

We now show a complete example of the application of these rules to explore the state space shown in Figure 4.8. Each node in the figure corresponds to a system configuration $\gamma \in \Gamma$. This state space contains a setup prefix with three states a, b and c, and four independent execution traces. Once combined with the never claim process, these execution traces result in graphs beginning on states d, e, f and g. In the figure, these graphs are shown unrolled as trees. We assume that the $next(\gamma)$ function returns the states in order, from left to right. Although the result is a single exploration trace, formed using the $\longmapsto_{ver'}$ transition relation, we will split it into several parts to discuss each one separately.

The exploration of the first execution trace is as follows. The first the transitions are $\overset{expl}{\longmapsto}_{ver'}$ that delve deeper into the state space. When node h is reached, it has d as it only child. Since that state has already been visited, a first $\overset{back}{\longmapsto}_{ver'}$ transition removes it from the corresponding continuation list (`Back2`). After that, since no states are left in the continuation, the exploration backtracks to the previous state (`Back3`). Then state i is visited, but the exploration backtracks again in the next step (`Back2`), since state i has no continuations. The same happens with state j. Since node d has no more continuations left, the exploration backtracks yet again, but this time from a state that does not belong in the setup prefix to one that does, triggering a $\overset{\textit{trace-end}}{\longmapsto}_{ver'}$ transition (`Trace-end`). This

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

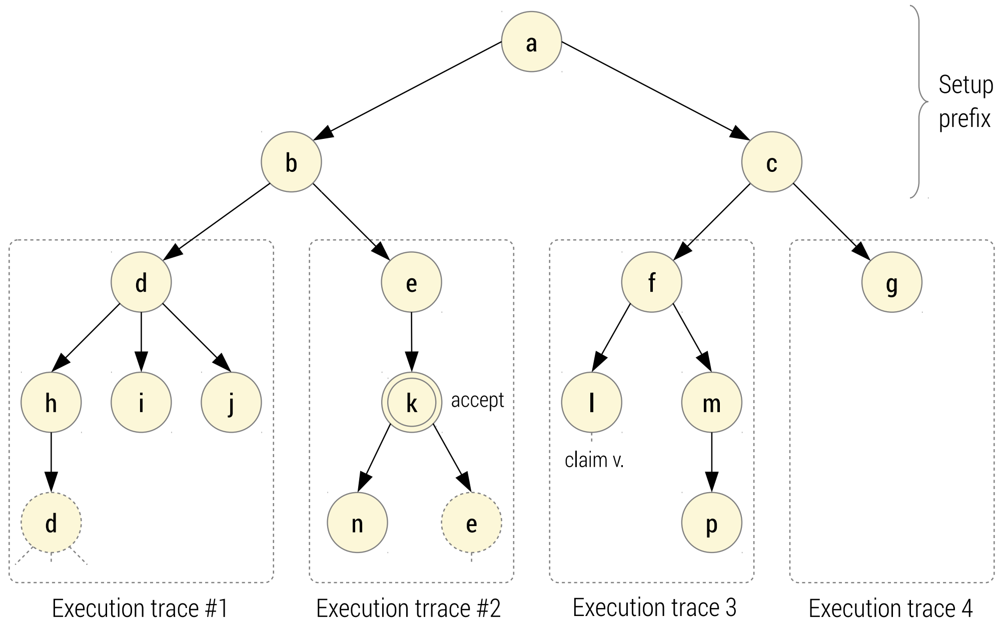


Figure 4.8: Trace exploration example

transition includes the prefix of the trace tree that was just explored completely, without any conclusive result: $[a,b,d]$.

		$\langle [a], [[b,c]], \{a\} \rangle$
`Expl`	$\overset{expl}{\longmapsto}_{ver'}$	$\langle [a,b], [[c],[d,e]], \{a,b\} \rangle$
`Expl`	$\overset{expl}{\longmapsto}_{ver'}$	$\langle [a,b,d], [[c],[e],[h,i,j]], \{a,b,d\} \rangle$
`Expl`	$\overset{expl}{\longmapsto}_{ver'}$	$\langle [a,b,d,h], [[c],[e],[i,j],[d]], \{a,b,d,h\} \rangle$
`Back2`	$\overset{back}{\longmapsto}_{ver'}$	$\langle [a,b,d,h], [[c],[e],[i,j],[]], \{a,b,d,h\} \rangle$
`Back3`	$\overset{back}{\longmapsto}_{ver'}$	$\langle [a,b,d], [[c],[e],[i,j]], \{a,b,d,h\} \rangle$
`Expl`	$\overset{expl}{\longmapsto}_{ver'}$	$\langle [a,b,d,i], [[c],[e],[j],[]], \{a,b,d,h,i\} \rangle$
`Back3`	$\overset{back}{\longmapsto}_{ver'}$	$\langle [a,b,d], [[c],[e],[j]], \{a,b,d,h,i\} \rangle$
`Expl`	$\overset{expl}{\longmapsto}_{ver'}$	$\langle [a,b,d,j], [[c],[e],[],[]], \{a,b,d,h,i,j\} \rangle$
`Back3`	$\overset{expl}{\longmapsto}_{ver'}$	$\langle [a,b,d], [[c],[e],[]], \{a,b,d,h,i,j\} \rangle$
`Trace-end`	$\overset{trace\text{-}end^{[a,b,d]}}{\longmapsto}_{ver'}$	$\langle [a,b], [[c],[e]], \{a,b,d,h,i,j\} \rangle$

The exploration trace then continues with the next continues with the next execution trace. After reaching state n and backtracking to k, rule `Accept` finds that the next continuation (state e) would yield a cycle with an accepting state within (state k). The

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

corresponding $\overset{accept}{\longmapsto}_{ver'}$ transition backtracks outside of the current trace (to state b), and includes the trace with the accepting cycle as part of its label: $[a,b,e,k,e]$.

...	...	$\langle [a,b],[[c],[e]],\{a,b,d,h,i,j\}\rangle$
Expl	$\overset{expl}{\longmapsto}_{ver'}$	$\langle [a,b,e],[[c],[],[k]],\{a,b,d,h,i,j,e\}\rangle$
Expl	$\overset{expl}{\longmapsto}_{ver'}$	$\langle [a,b,e,k],[[c],[],[],[n,e]],\{a,b,d,h,i,j,e\}\rangle$
Expl	$\overset{expl}{\longmapsto}_{ver'}$	$\langle [a,b,e,k,n],[[c],[],[],[e],[]],\{a,b,d,h,i,j,e,n\}\rangle$
Back3	$\overset{back}{\longmapsto}_{ver'}$	$\langle [a,b,e,k],[[c],[],[],[e]],\{a,b,d,h,i,j,e,n\}\rangle$
Accept	$\overset{accept^{[a,b,e,k,e]}}{\longmapsto}_{ver'}$	$\langle [a,b],[[c],[]],\{a,b,d,h,i,j,e,n\}\rangle$

The exploration backtracks back to the root state a, and then to the next continuation: state c. The third trace contains a never claim violation that is detected by rule `Claim` from state f. From this state, function *next* returns *claim_v* for the first continuation l. The transition includes the offending trace as part of its label: $[a,c,f,l]$. States m and p are not visited.

...	...	$\langle [a,b],[[c],[]],\{a,b,d,h,i,j,e,n\}\rangle$
Back3	$\overset{back}{\longmapsto}_{ver'}$	$\langle [a],[[c]],\{a,b,d,h,i,j,e,n\}\rangle$
Expl	$\overset{expl}{\longmapsto}_{ver'}$	$\langle [a,c],[[],[f,g]],\{a,b,d,h,i,j,e,n,c\}\rangle$
Expl	$\overset{expl}{\longmapsto}_{ver'}$	$\langle [a,c,f],[[],[g],[l]],\{a,b,d,h,i,j,e,n,c,f\}\rangle\rangle$
Claim	$\overset{clam_v^{[a,c,f,l]}}{\longmapsto}_{ver'}$	$\langle [a,c],[[],[g]],\{a,b,d,h,i,j,e,n,c,f,l\}\rangle$

The final execution trace contains a single state g without continuations. After that state is visited, the next transition jumps out of the trace and is labeled accordingly as explored without property violations.

...	...	$\langle [a,c],[[],[g]],\{a,b,d,h,i,j,e,n,c,f,l\}\rangle$
Expl	$\overset{expl}{\longmapsto}_{ver'}$	$\langle [a,c,g],[[],[],[]],\{a,b,d,h,i,j,e,n,c,f,l,g\}\rangle$
Trace-end	$\overset{trace\text{-}end^{[a,c,g]}}{\longmapsto}_{ver'}$	$\langle [a,c],[[],[]],\{a,b,d,h,i,j,e,n,c,f,l,g\}\rangle$
Back3	$\overset{back}{\longmapsto}_{ver'}$	$\langle [a],[[]],\{a,b,d,h,i,j,e,n,c,f,l,g\}\rangle$
End	$\overset{end}{\longmapsto}_{ver'}$	*stop*

After backtracking and reaching the root state again, the exploration ends as there are no more continuations left to visit.

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

Part III

Applications

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

Chapter 5

Analysis of ns-2 network simulations

The first application of OptySim concerns the analysis and optimization of communications network scenarios. As is custom in the networks community, we use a network simulator as the system under test, which allows easier integration with our framework. In particular, we use the well-known ns-2 network simulator [11].

A common application of network simulations during development is parameterized analysis, e.g. for performance optimization. Automating the generation of parameter configurations and the simulation itself is vital for large-scale, repeatable analysis. Furthermore, the objective of the analysis can be taken advantage of for carrying out more meaningful and directed studies.

In this chapter we first describe the architecture and function of ns-2, and discuss what we consider an execution trace in the context of ns-2 network simulations. The execution trace is extracted using a generic C++ library (which can be reused for integrating with other systems) and a specific library that integrates with ns-2. These libraries provide an implementation of the communication protocol, extraction of system states after each relevant simulation step, and other required pieces.

We present two different network scenarios as case studies: video streaming over TCP on a mobile environment, and objective VoIP call quality evaluation. In the first scenario we focus on analyzing reliability requirements on the system models, and on optimizing the performance of the streaming over a mobile link with predictable losses. For the second scenario we developed an improved version of a existing quality of experience (QoE) evaluation model [39]. Then, we set up a validation framework that checks the results of new evaluation models against a reference model.

5.1 The ns-2 network simulator

The ns-2 network simulator [11] has been one of the most used network simulators in academia over the past few years. At its core, ns-2 is a discrete event simulator, i.e.

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

events are queued and dispatched according to their trigger time. In this section we give a brief tour of how ns-2 simulations are carried out, and what are the possible sources for extracting execution traces.

5.1.1 Architecture

ns-2 is implemented in a mixture of C++ and OTcl code. C++ is mainly used in the simulator core, network protocols and other elements that need to be efficient. OTcl is used in places where speed can be traded for flexibility, e.g. for setting up simulation scenarios. Most of the C++ class hierarchy can be accessed from OTcl code through a "mirrored" class hierarchy, which enables writing OTcl code has access to the whole simulator system.

The key elements of the architecture of ns-2 are nodes, links, agents, applications and packets. Nodes are the basic element of a network topology. A node can be anything from a computer to a mobile device or a router. Nodes are connected through links, which are abstractions of real access technologies. Links are responsible for packet queuing, delays and errors. Each node is usually composed of one or more agents and applications. Agents are endpoints for network-layer packets. Typically, an agent represents a network protocol or part of one, such as TCP or UDP. Applications sit on top of agents and may be used as traffic generators and sinks. Packets are the basic unit of information exchanged between these entities. As with real network packets, ns-2 packets may include agent- or application-specific headers and data.

While the architecture of ns-2 is quite generic, existing modules do not use it in the same way. For instance, packets do not usually carry application data, and even when they do, many transport layer protocols do not forward the data to the corresponding application. Instead, the application is just notified of how many bytes of data were received.

As an event-driven simulator, the core of ns-2 is the scheduling algorithm. The tool includes four scheduler implementations which differ in the underlying data structures, but which should yield exactly the same scheduling. The scheduler runs iteratively, selecting the next earliest event, executing it to completion and then returning to select the next event. Only one event can be executed at a given time and the order of dispatch of simultaneous is deterministic, i.e. events are dispatched according to their firing times and events scheduled for the same time instant will be dispatched in the same order they were scheduled. The basic scheduling algorithm is outlined in Listing 5.1.

Each event has a handler associated with it, i.e. the object responsible for attending the event. For instance, if a packet is sent over a link, it will be scheduled some time in the future (e.g. taking into account the time it would take to traverse the link) and the destination node will be its handler. When this event is fired, the node will be handed the event with the packet and process it accordingly. A timer, e.g. for queue management or

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

```
void Scheduler::run() {
    Event* e = 0;
    // While there are more events
    while (!halted && (e = dequeue())) {
        // Dispatch the next event
        dispatch(e);
    }
}

void Scheduler::dispatch(Event* e) {
    // Advance simulation time to that of the current event
    clock_ = e->time_;
    // Execute the event with the designated handler
    e->handler_->handle(e);
}
```

Listing 5.1: ns-2 basic scheduling algorithm implementation

generating traffic, works in the same way by scheduling an event in the future. Event handlers are run to completion and cannot be interrupted by the scheduler.

Figure 5.1 shows an example where several events are scheduled along the timeline. The scheduler selects them in order and calls their handlers, which may schedule new events in the future. In this example, the first event handled also schedules an event in the future. Then the clock advances to the next available event and the process is repeated.

Regarding packet exchanges, it is worth noting that not all of them are scheduled in this way. For instance, agents and applications typically forward packets directly within the same node, without scheduling an event, calling the `recv` method on the receiver. This means that these packets are delivered and processed instantly without delay in the same simulated time instant. It may be the case that, once a packet as been received at a node, it is processed by the whole protocol stack in this way in the same time instant. Is is usually trivial, however, to delay a packet sent to an agent. If a packet is scheduled with an agent set as its handler, the `handle` method will be eventually called, which by default calls the method `recv`, assuming the given `Event` is a `Packet`. Thus, agents usually need to implement only the `recv` method.

The event scheduling algorithm is deterministic. However, the may be probabilities involved when deciding the time in the future in which to schedule an event. For instance, a traffic generator may use a probability distribution to decide the time in which the next packet will be sent. ns-2 provides a random number generator, which server as the basis to implement random variables that provide several parameterized probability distributions, including normal, exponential and Pareto.

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

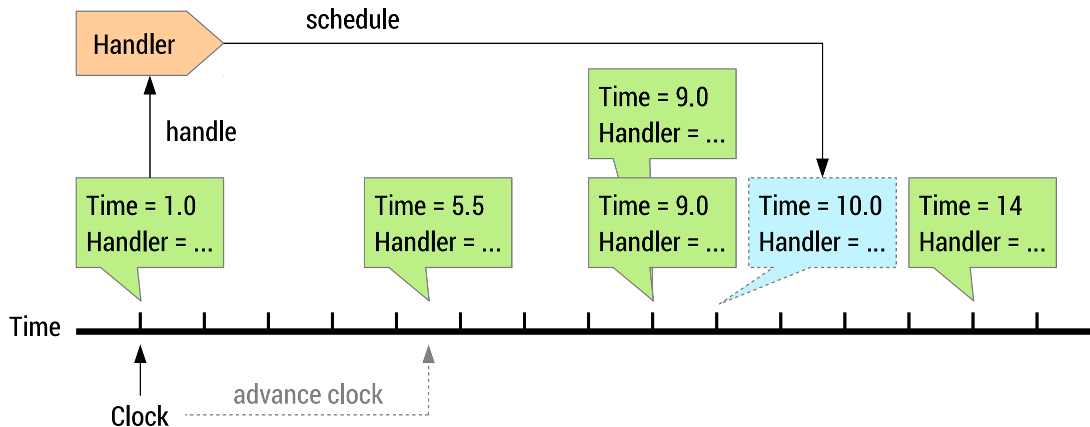


Figure 5.1: Event scheduling and handling example in ns-2

5.1.2 State space and traces

Given a random seed, a ns-2 simulation produces a deterministic execution trace, i.e. a fixed sequence of states. Once a simulation scenario is set up, the simulation itself is carried out by the scheduler algorithm, and the corresponding events and handlers.

The complete definition of a trace depends on the definition of state. Each state is composed of the global event list, the current simulation time and the state of every network element in the scenario. The boundary between two states can be defined with different levels of granularity. For instance, a new state could be produced by each sentence executed in an event handler. However, the number of states generated in this way could be too high to be practical for many purposes. Another possible definition of state could be at the event level, i.e. a new state is produced after each event is processed by an event handler. However, we argue that such a fine level is often not required by the analysis, and that the programmer or the analysis software may decide where to set the boundaries, e.g. by monitoring relevant events or variables. As we will discuss in the following section, these are the same ideas behind our trace projections (see Chapter 4). Figure 5.2 shows the evolution of states at these levels of granularity.

ns-2 supports two main procedures for collecting information from a running simulation: variable tracing and packet tracing. The former enables the recording of changes in variables of interest, while the latter may be used to monitor each packet that goes through a queue or link.

Variable tracing is enabled by using special wrapper classes instead of the regular data types, such as `int` or `double`, for the variables of interest. By overloading the basic operators (e.g. $+$ or $*$) these wrapper classes can notify a `Tracer` object when the underlying value changes. This object may act on these changes in any useful way, such as writing to a trace file or performing other actions.

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

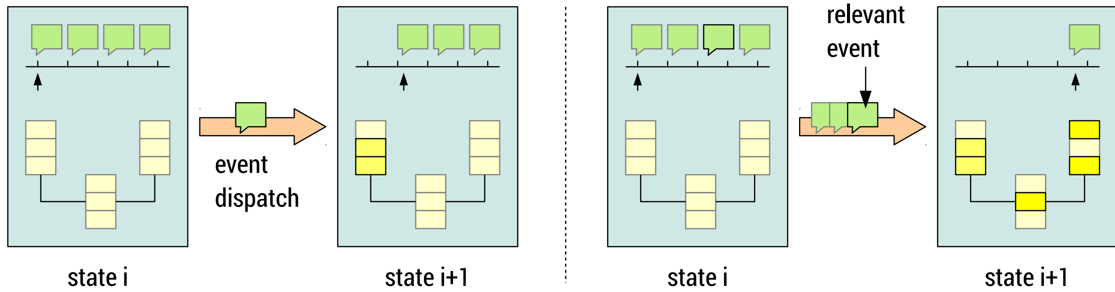


Figure 5.2: ns-2 trace: new state after each event vs. after a relevant event

On the other hand, packet tracing is a transparent mechanism to record packet-related events such as packet arrivals and departures, or packets dropped at queue or link. Packet tracing is achieved through `Trace` objects, which is a subclass of `Connector` that can be inserted between two objects.

In addition, many variables can be accessed from OTcl code without modifying the C++ source code to perform additional monitoring. Using the C++/OTcl binding mechanism C++ members can be accessed or modified from OTcl. Many parameters of existing agents are already bound for convenience using this mechanism.

5.2 Extraction of ns-2 execution traces

In order to analyze a ns-2 simulation, a suitable execution trace must be provided. As described in the previous section, ns-2 supports two main sources from which to extract a view of a system execution. Perhaps, the most widely used source in the networks community is tracing packets at the link level. However, in our case studies we are more interested in the internal state of some of the network elements, whose updates can be traced by ns-2, as well as in aggregated statistics.

As explained in Chapter 4, it is not usually feasible to analyze a full execution trace. The previous section describes the elements that are present in a ns-2 simulation, which can grow quickly with the complexity of a network scenario. In addition, the number of events impacts directly the length of an execution trace. For our integration with ns-2 we chose to apply a counter projection (see Section 4.2.1). We support using the simulation time as the counter (instead of a fresh variable incremented by one per projected state) since the time is also a strictly increasing variable between states, and is often included in the projection.

In addition to identifying the elements of the trace, a front-end module must be developed in order to communicate with the analyzer. This front-end should handle the protocol described in Chapter 3, and help with extracting and projecting the trace.

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

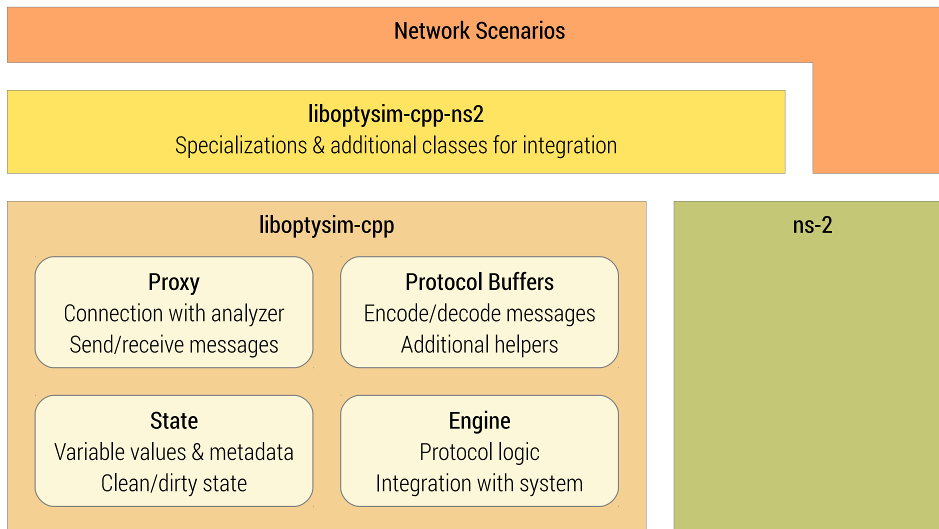


Figure 5.3: Components of ns-2 integration

In this section we describe the software modules we developed in order to communicate with the analyzer and provide a suitable abstract execution trace.

5.2.1 Overview of integration with ns-2

Figure 5.3 shows an overview of the components involved in the integration with ns-2. On the bottom left corner there is the liboptysim-cpp library. This library was developed in C++ to serve as a starting point for developing libraries tailored for specific systems which supported this language. The most important capabilities offered by this library are communicating with the analyzer through the protocol described in Chapter 3, and keeping track of the current state of the system for applying the counter projection described in Chapter 4.

In order to integrate with ns-2, some additional classes were provided in a separate library: liboptysim-cpp-ns2. For instance, this library provides a `Scheduler` implementation that identifies simulation states as described in Section 5.1 and reports them to the analyzer if appropriate.

The network scenarios to be executed and analyzed are written as regular ns-2 scenarios, but with access to certain features provided by this integration library. For instance, measure reports may be triggered from the scenario, e.g. after a relevant event. Although scenarios can be largely be executed without any modification, some changes are still necessary. For instance, the scenario must be adapted to the way parameters are provided from the analyzer.

In the remainder of this section, we will discuss these components with more detail.

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

5.2.2 Generic C++ library

The generic part of the integration with C++ systems is located in a library called liboptysim-cpp. As shown in Figure 5.3, the main responsibilities of this class are: handling protocol buffer messages, communicating with the analyzer and keeping track of the state of the system. Finally, the “engine” is the core of the logic implemented in the front-end library. Figure 5.4 shows the main classes of this library, grouping them according to these responsibilities. The elements painted with a darker shade are part of external libraries.

The implementation of the communication protocol (see Chapter 3) is distributed across these components. Proxy classes receive and send the messages, while the messages themselves are encoded and decoded using the protocol buffers classes. Received messages are handled in the engine component. This component is also responsible for initiating a communication with the analyzer when appropriate, e.g. sending a *Report* when the tracked variables of the system change.

The counter projection (see Chapter 4) is mainly handled in the state component. This component tracks the variables of interest (as indicated by the analyzer during the setup phase), and notifies the engine when one of them changes. The engine interprets this as a state that should be part of the projected trace, and sends the corresponding *Report* to the analyzer.

The rest of this section expands the description of each of these components, and ends with an example of how the protocol is implemented through them.

Protocol Buffers

Protocol buffer messages are mostly handled by C++ classes generated automatically from the definitions. There is one class per message type, which extend and use classes from the C++ library for protocol buffers, libprotobuf. The full code generation process is explained in Section 3.4.4.

In addition, the `ProtobufHelper` class provides additional functions for handling the messages. In particular, it provides several static methods for reading and writing messages with a corresponding *Header* message. In the protocol, every message must be preceded by a *Header* message containing both the type and length in bytes of the following message. This fixed length message helps identifying and delimiting the messages sent through a stream.

Proxy

The `Proxy` class is the main entry point for communicating with the analyzer. It contains methods for sending any of the required messages in the protocol, and may notify of arriving messages to a registered `ProxyListener`. Some of the methods in `Proxy` are

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

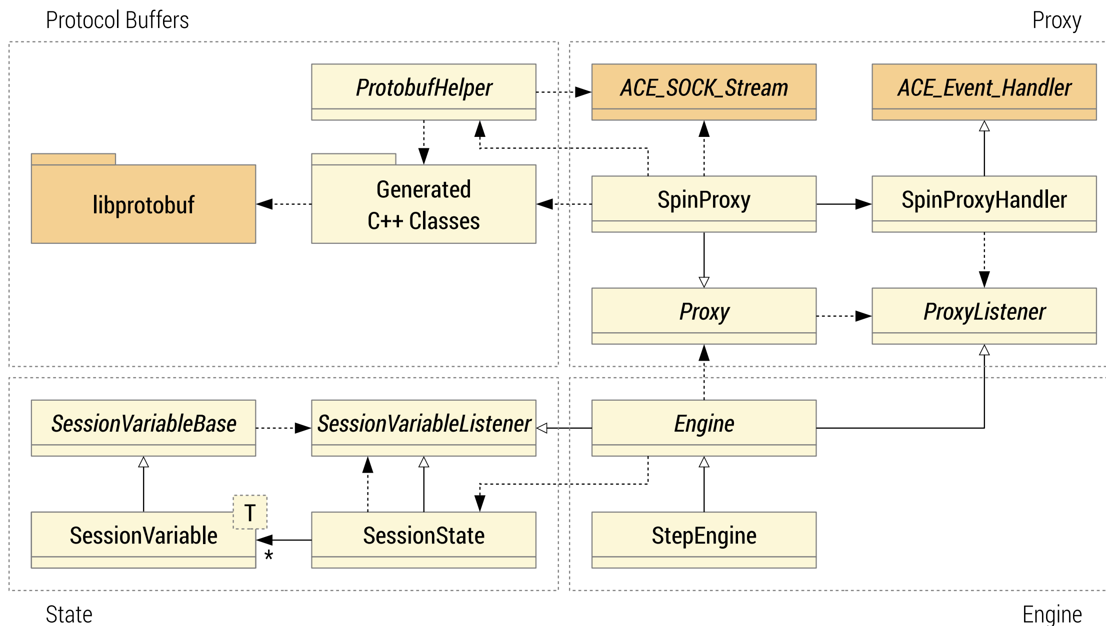


Figure 5.4: Main classes of liboptysim-cpp

pure virtual, and are meant to be implemented by a subclass using a specific means of communication. A subclass, `ProxyStub`, was developed for testing purposes.

Our main `Proxy` implementation is `SpinProxy`, which uses ACE sockets and reactors [107][108]. Messages are serialized and sent through the socket using `ProtobufHelper`. To receive messages, an event handler, `SpinProxyHandler`, is installed in a global reactor provided by the library. The reactor pattern [106] is used to decouple the event sources from their handlers. Received messages are decoded and relayed to a registered `ProxyListener`, most likely an `Engine` instance.

State

A fundamental part of the library is providing an execution trace by tracking the individual states of the system. Typically, only a subset of the system variables are made available to the analyzer, a so-called abstract state (see Chapter 4). `SessionState` holds the current values of this subset of variables. The values themselves (and additional meta-data) are stored in `SessionVariable` instances. These classes are not intended to replace the regular variables in a C++ system, although they could be used this way. System integration libraries should provide mechanisms to update the `SessionVariable` instances when the system variables change. These changes are notified to the `SessionState`, which will mark the state of the system as “dirty”. At the same time, a `SessionVariableListener` registered within `SessionState`, usually an `Engine` instance, may be notified of the

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

changes. This enables the process of projecting an execution trace if the analyzer is only interested in states where one of the relevant variables changes.

Engine

The `Engine` class and subclasses centralize a significant portion of the logic behind the protocol, the state abstraction and system integration. There should be only one `Engine` instance in the system. On one hand, this `Engine` instance is notified of incoming messages and changes in system variables through the implemented `ProxyListener` and `SessionVariableListener` interfaces, respectively. The default implementation handles some of these events, e.g. when the parameter values are received from the analyzer, the corresponding `SessionVariable` instances are updated with the provided values. On the other hand, the `Engine` should decide when to send reports and handle some commands. However, the base class does not provide any implementation for this, but rather lets the subclasses provide a suitable logic.

For our case studies, it made sense to have some notion of "atomic step": an extension of system execution which is observed as a single unit by the library, and which is not interrupted. For instance, in a ns-2 network simulation, an event dispatch and the associated handling can be considered as a step. Several variable changes are collapsed into a single state change, and command processing is deferred until the step is finished. Thus, we provided a `StepEngine` which is prepared for these kinds of systems. Other systems may require different implementations, e.g. an `Engine` subclass which reacts in real time to commands and changes in the system.

The `StepEngine` class provides a single `step()` method which should be invoked after each system step. Invoking this method is left to the integration libraries, like the ns-2 one described in Section 5.2.3. The `step()` method first checks the commands sent from the analyzer, which are stored in a queue during the system execution, and then the state of the system state. If appropriate, it will trigger a measure report for the analyzer.

Example

Figure 5.5 shows a sequence diagram with an example of the implementation of the communication protocol across several classes. This diagram is partial: it only shows the second half of the setup phase, and the main loop of the execution phase. Furthermore, some of the interactions have been simplified for the sake of illustration. For instance, some of the behavior of the `SpinProxy` is really implemented in its parent class, `Proxy`, or in a `SpinProxyHandler` instance.

`SpinProxy` is tasked with receiving and forwarding the messages received from the analyzer. The *VariableSelection* message triggers a remapping of the variables tracked in `SessionState`, while the *ParameterSetup* message is handled in the `StepEngine` by

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

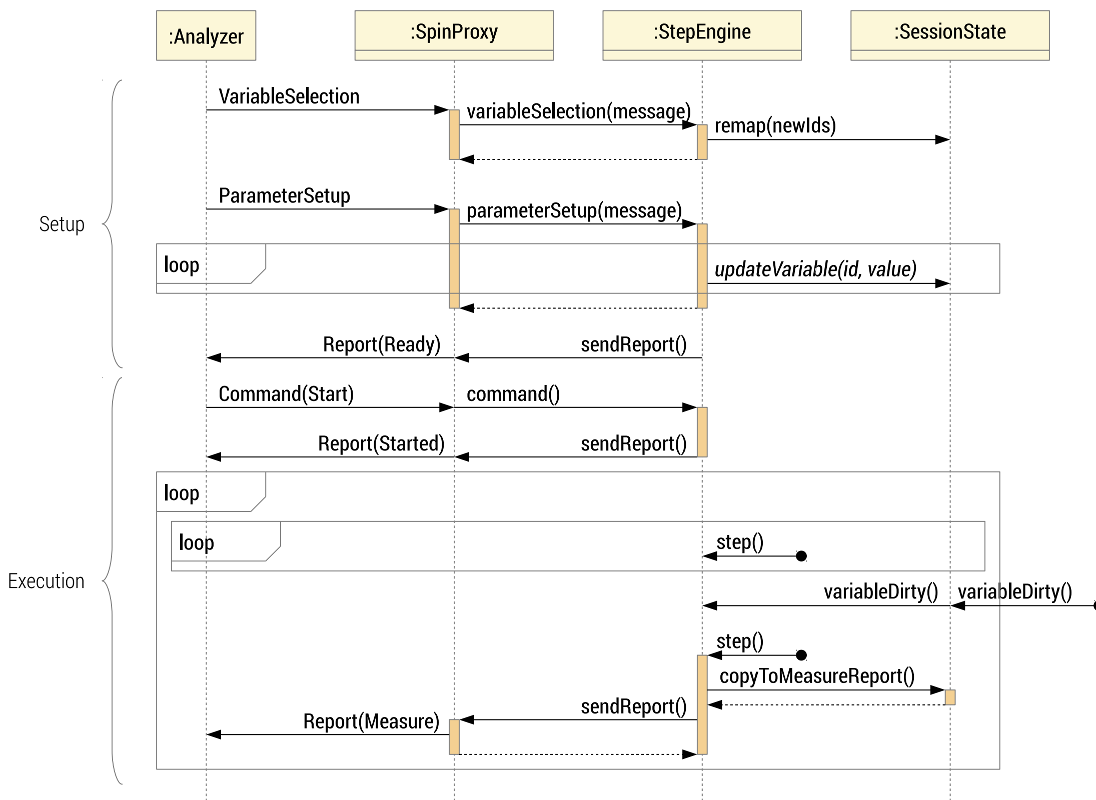


Figure 5.5: Partial sequence diagram of protocol implementation

updating the current value the given parameters (here, `updateVariable(id, value)` groups several different calls).

During the execution phase the `StepEngine::step() method` is called repeatedly, once for each system step. Since the variables tracked have not changed during any of these steps, no report is sent to the analyzer. At one point, one (or more) of these variables is updated during one of the system steps. The `SessionState` is notified, which in turn notifies the `StepEngine`. This time, during the next `step()`, a *Report* of type `Measure` is built and sent to the analyzer. This loop will continue until either the system ends, or the analyzer instructs the system to stop.

5.2.3 ns-2 integration library

We developed a specific library for ns-2 to bridge the gap between the generic C++ library and the network simulator. Figure 5.6 shows the main classes of this library. Again, the elements painted with a darker shade are external to our library.

First, the integration library provides the `Ns2StepEngine` class, a subclass of `StepEngine`. The only difference with the latter is that `Ns2StepEngine` sets the received

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

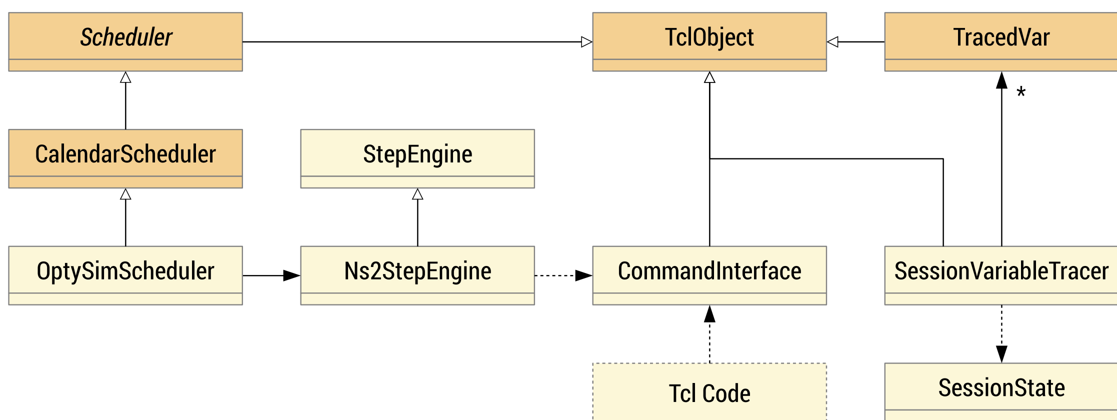


Figure 5.6: Main classes of liboptysim-cpp-ns2

parameters as global OTcl parameters in the simulation scenario. This mirrors a pattern observed in many existing network scenarios for setting parameters, and reduces the friction when using the integration library.

As described in Section 5.1, the scheduler is responsible for making the simulation advance. Each event dispatch can be considered as generating a new state in the simulation. Thus, it seems appropriate to call the `step()` method on the engine after handling an event. For this we developed a extended the `CalendarScheduler` class (the default scheduler implementation), altering the `run()` method to include this call. This would be inserted right after line 6 in Figure 5.1.

The library also provides some functionality that can be used from the network scenarios written in OTcl. These range from statistic functions to triggering measure reports. In order to enable the latter, the `CommandInterface` class was developed, which accepts several OTcl commands to perform these functions. A global instance of this class is available in OTcl for every simulation. This class also provides static methods for running OTcl commands in the interpreter provided by ns-2, should that be necessary.

ns-2 provides variable tracing in order to be notified when a variable changes (see Section 5.1.2), which the integration library supports partially. `TracedVar` instances can be traced using a `SessionVariableTracer`, which also associates each with a `SessionVariable<T>`. `SessionVariableTracer` associates a `TracedVar` with a `SessionVariableTracer`, and registers itself as a tracer of the former. Whenever one of these `TracedVar` instances changes its value, the corresponding `SessionVariable<T>` is updated, possibly triggering a measure report after completing the current event.

Using the commands offered by `CommandInterface`, it is also possible to notify of changes in variables of interest which are not traced. This may be useful for variables that are not stored but rather computed on-the-fly as needed, e.g. network statistics.

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

5.3 Case study: video streaming over TCP on mobile environments

The first case study for ns-2 involves streaming a video over TCP to a mobile phone [93]. We will also check the behavior of FreezeTCP [69], a TCP variant tailored for mobile environments. We will use this case study for performance and reliability analysis.

5.3.1 Network scenario

Many efforts have been directed towards defining an appropriate set of protocols for multimedia streaming in real time, such as RTP (Real-time Transport Protocol) [3]. Interestingly, one of the most popular video streaming websites, YouTube, uses plain HTTP over TCP for transmitting the multimedia content to the clients, just like any other web content. To deal with bandwidth limitations or variability, video playback does not start immediately as it could lead to frequent video stalling when running out of video frames to play. An initial playout buffer is employed to cache an initial portion of the video, long enough to hopefully avoid stalling. The size of this buffer can be fixed, but is usually adapted to the quality of the connection, so if the data connection is slow more video will be buffered before starting the playback.

New radio access technologies such as HSDPA (High-Speed Download Packet Access) [5] have given mobile phones enough bandwidth to consume video services effectively. However, these devices pose additional problems that are not present in regular wired networks. In addition to a smaller bandwidth capacity, wireless links are more prone to errors and disconnections due to changing medium conditions or cell handover, among others. These issues may affect upper transport layers such as TCP, which were not designed for this environment. For instance, a short disconnection may be misinterpreted by TCP as a sign of network congestion and thus activate anti-congestion measures, which will degrade performance. This has led to some protocol variants tailored for mobile environments [114].

FreezeTCP [69] is a variant of TCP which improves the response to predictable connection drops. One of the key features of FreezeTCP is that it takes advantage of existing TCP mechanisms, and only the client (the mobile device) has to be modified. When the client detects an impending disconnection, it warns the server in advance in order to “freeze” the connection state. When the connection is reestablished, the client notifies the server and the connection is “unfrozen”, returning to the same state as when the disconnection notification was sent, without polling or going through the “slow-start” phase of TCP. For FreezeTCP to work some information must be shared between TCP and the physical layer, so that disconnections can be predicted based on changes in the physical link. The time in advance to notify a disconnection is an important factor. If it is too soon, bandwidth may be unused for too long while it is “frozen”. On the

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

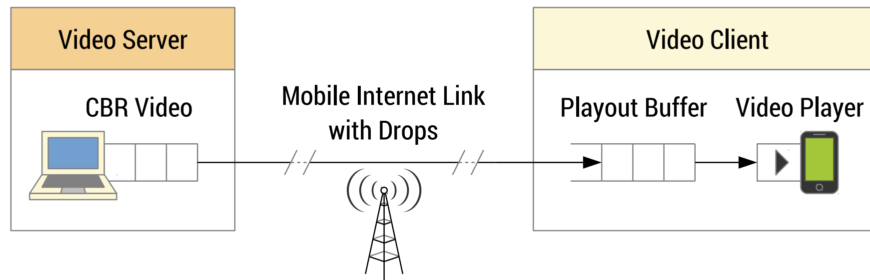


Figure 5.7: Overview of video download over TCP

other hand, if the server is warned too late, the notification may not reach its destination. Furthermore, while FreezeTCP can cope with long or frequent disconnections, it is not suitable for links with a high error rate.

Figure 5.7 shows a high level model of the scenario. The video server is a web server and the client is a regular web client, using the HTTP protocol over TCP. The client has a buffer where the video frames are stored. The playback process retrieves the video frames from this buffer, after an initial playout buffer has been filled. To simplify the model a constant frame size and rate is assumed.

The behavior of the client is shown on a state diagram in Figure 5.8. Initially, the buffer is empty and the playback has not started yet. When the client receives the first video frame it goes to the "buffering" state. Playback does not start until the playout buffer reaches a certain threshold, which takes the client to the "playing" state. If the client runs out of frames during playback, it must stop and go back to the "buffering" state, until the playout buffer is filled again. If the video is completely buffered, playback starts (if it was not playing already) and the video is said to be received, so the client moves to another state. Finally, the last state is reached when the video is completely played, which is guaranteed to happen eventually if the client reaches the "fully buffered & playing" state.

5.3.2 Reliability analysis

We used OptySim to debug our implementation of the TCP video client and server. We also tested and successfully fixed a couple of issues in the ns-2 implementation of FreezeTCP [69] available from [9].

In our scenario, the TCP video client is a new application, while the server is a CBR (constant bit rate) traffic generator. Listing 5.2 contains part of the video client code, in particular the methods where its status is manipulated. `recv()` is called when a new TCP packet arrives with video data, and `nextFrame()` is scheduled periodically when the video is playing. The former fills the playout buffer, from which frames are read by the latter.

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

```
void TcpVideoClientApp::recv(int nbytes) {
    bytes += nbytes;
    buffered_bytes += nbytes;
    if (bytes >= video_size) {
        status = RECEIVED;
    }
    else if (status == STOPPED && buffered_bytes < buffer_size) {
        status = BUFFERING;
    }
    else if (status == STOPPED || status == BUFFERING
            && buffered_bytes >= buffer_size) {
        status = PLAYING;
        timer.sched(delay);
    }
}

void TcpVideoClientApp::nextFrame() {
    if (buffered_bytes >= frame_size) {
        buffered_bytes -= frame_size;
        video_size_played += frame_size;
        if (video_size_played < video_size) {
            timer.resched(delay);
        }
        else {
            status = FINISHED;
        }
    }
    else if (buffered_bytes < frame_size && status == RECEIVED) {
        buffered_bytes = 0;
        status = FINISHED;
    }
    else {
        status = STOPPED;
    }
}
```

Listing 5.2: Part of video client implementation in ns-2

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

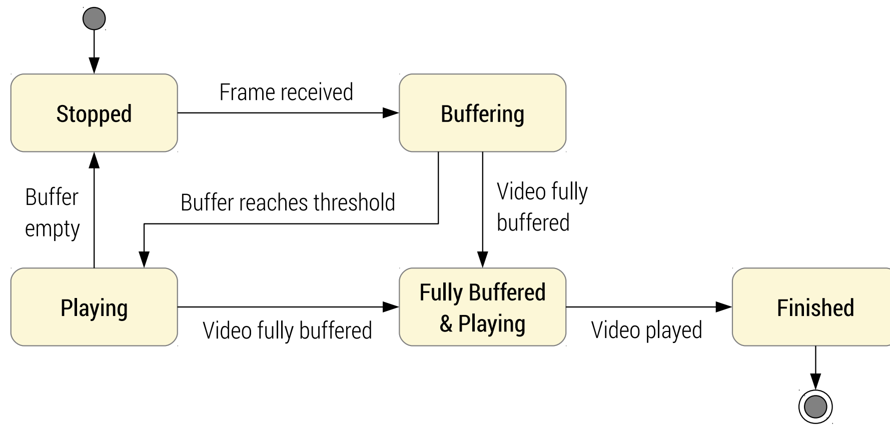


Figure 5.8: State diagram of the video client

One of the conditions that can be tested is whether the video client finishes successfully when it has already buffered the whole video. We can check that condition with a set of automatically generated parameter configurations and the following objective:

$$\begin{aligned} ltlAccept: &\quad \Diamond\,(received\ \wedge\ \Diamond finished) \\ def: &\quad received\ :=\ status = 3 \\ def: &\quad finished\ :=\ status = 4 \end{aligned} \tag{5.1}$$

In order to execute this test, we must select the *status* variable to be part of the projected trace (along with the time, which is selected by default) so that it can be used in the LTL formula. This means that the projected trace will only contain this variable, and that only the states in which *status* changes will be projected. This reduces both the size of the state and length of the trace dramatically.

All simulations pass this test. We can strengthen it by adding more information about the state in which the video client must be at the end, e.g., the playback buffer must be empty:

$$\begin{aligned} ltlAccept: &\quad \Diamond\,(received\ \wedge\ \Diamond reallyFinished) \\ def: &\quad reallyFinished\ :=\ status = 4\ \wedge\ bufferedBytes = 0 \end{aligned} \tag{5.2}$$

The projected trace must also contain the *bufferedBytes* variable for this test. This time no simulations are returned tagged with *accepted*, meaning that none of them satisfied this strengthened condition. If video playback finishes normally, it means that the buffer must have enough frames to play it completely, so it may be that too much data is sent to the client. This can be confirmed adding the following to the previous objective:

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

```
[measures] -----------------------------
time     status  video_size  video_size_played  buffered_bytes
0.0000   0       1024000     0                  0
0.3233   2       1024000     0                  1064
0.8331   1       1024000     0                  10640
22.0752  3       1024000     652288             372344
34.1665  4       1024000     1024000            40000
[REJECT] <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
Reject at 34.1665: now.status == 4 && now.buffered_bytes != 0
[finish] -------------------------------
Simulation running time=0.000076977s
```

Figure 5.9: Sample trace output for reliability objective violation

$$reject: \quad status = 4 \wedge bufferedBytes > 0 \tag{5.3}$$

This time, all simulations are marked as rejected because of this addition. Figure 5.9 shows part of the output generated by OptySim when analyzing one of these simulations. It contains the values of the measured variables for each state, and the objective that triggered the end of the analysis of this simulation. In this scenario, events are sent only when the status of the client changes, which explains the relatively low number of states which the trace is made of. In this trace, there are 40,000 bytes left in the buffer after the client has played the entire video. This problem is caused by ns-2 incorrectly reporting the TCP header (40 bytes) as part of the payload notified to the application layer. As a workaround we can instruct the TCP server not to include the TCP headers in the packet itself. However, the header of the SYN packet used to initiate the TCP connection is reported as well, so in addition we must ignore the first packet received by the video client. This condition should be added at the beginning of `recv()`.

We prepared another set of scenarios for testing the FreezeTCP implementation available from [9]. One of the tests forces the client side of the protocol to freeze or unfreeze at fixed times, trying several combinations of these commands. Using no additional objectives, we discovered an issue with the handling of an “unfreeze” command before the first “freeze” command, due to the fact that it did not have appropriate default values for some state variables. We also observed that the throughput of some simulations did not recover after unfreezing. Using another objective we checked whether the saved state was restored properly after unfreezing. We discovered that the FreezeTCP model did not handle two consecutive “freeze” commands correctly. This may be the case if the client side sends more than one warning message to ensure that at least one of them reaches the server side, as suggested in [69]. This problem can be solved by adding a variable

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

to hold the state (unfrozen or frozen) of the protocol, and reacting only to the first of a series of “freeze” commands.

It is worth noting that some of the tests, including our second example, require the simulation models to be instrumented in order to access some of their internal state. For instance, we instrumented the FreezeTCP code in order to read the frozen state of the protocol and be notified whenever a “freeze” or “unfreeze” message reaches the server side. Other variables, such as the internal state of the generic TCP implementation, were already available to us through the OTcl binding mechanism.

5.3.3 Performance analysis

We use the scenario described above to study the performance of the video playback in the presence of connection drops, the influence of the playback buffer and the effectiveness of the FreezeTCP variant. This scenario is similar to the one studied in [110], which also uses ns-2 to simulate a set of configurations.

The scenario is composed of two nodes connected through a link which drops periodically for a given amount of time, to simulate network disconnections. Each node has a TCP agent with an application on top: an `Application/Traffic/CBR` instance and an `Application/TCPVideo/Client` instance, respectively. The latter was developed in C++ to simulate both the reception of video using TCP and the playback engine at the client. The client has a playout buffer (a simple counter) in which all TCP segments are placed upon arrival. This buffer is read 30 times per second to simulate the fetching of each video frame by the playback engine at 30 fps. The client also makes its state available through a traceable instance variable.

The parameters and their possible values are defined in Table 5.1. These parameters are related to both the environment, e.g. the duration of uptime and downtime periods of the link, and the elements of the protocol stack, e.g. segment size and maximum window size for TCP. The combination of the different valuations of each parameter constitutes the parameter space that we want to explore.

We are interested in those parameter configurations in which the video playback is smooth and does not stop for rebuffering once started. In our tool we allow the definition of more than one objective, stating if it is a desired objective (*accept*) or something that should not happen (*reject*). Thus, for this experiment we would write the following two objectives:

$$
\begin{aligned}
accept:&\quad status = 3\\
ltlReject:&\quad \Diamond\,(playing \wedge \Diamond\, stopped)\\
def:&\quad playing \;:=\; status = 1\\
def:&\quad stopped \;:=\; status = 0
\end{aligned}
\tag{5.4}
$$

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

Variable	Values
Link uptime	10 s
Link downtime	100 ms, 300 ms, 600 ms, 1 s
Link delay	100 ms, 125 ms, 150 ms
Link bandwidth	288 Kb, 320 Kb, 384 Kb
Max. TCP window	5 KB – 10 KB, 1 KB increments
TCP segment size	0.1 KB – 0.3 KB, 0.05 KB increments
Playout buffer length	1 s – 20 s, 1 s increments

Table 5.1: Scenario parameters and their possible values for the case study

The first objective declares that, in order to be considered successful, the *status* variable of the video client should reach value 3, i.e., the “finished” state. The second objective is an LTL formula used to reject those simulations where once the client reaches the “playing” state it should not reach the “stopped” state in the future. Both *playing* and *stopped* are defined in terms of the *status* variable.

In addition, we want to use the following search optimization objective. The satisfiability of the previous objective depends on the size of the playout buffer, among other variables. If a certain parameter configuration results in a simulation that meets the objective, then increasing the size of the playout buffer while keeping the rest of the parameters unchanged will have no negative effect on the objective. That is, these simulations can be safely skipped, but they should be returned as part of the parameter configurations that meet the objective. This can result in significant time savings if such deductions can be described and are likely to happen in the scenario generation and simulation process. This search optimization objective can be written as:

$$acceptSimilar: \quad bufferLength > bufferLength' \wedge \Big(\bigwedge_{p \neq bufferLength} p = p' \Big) \tag{5.5}$$

The purpose of this formula is to match the current parameter configuration with a similar parameter configuration that has already been run and that meets the objectives. If one of the previous parameter configurations that has been accepted has the same values for all the variables except for a smaller buffer length, it will match the formula and the current parameter configuration will be accepted without being actually run.

Once our objectives have been decided, it is clear that our projected trace only requires the *status* variable, for both the *accept* and *ltlReject* objectives. The search optimization operates over the parameters of the simulation, which are always stored for each simulation. Therefore, *bufferLength* and the rest of the parameters do not have to be explicitly selected.

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

		Parameter configurations				Counter projection		
Objective	Search opt.	Generated	Accepted	Rejected	Not exec.	States	S. size	Time
		21 600	–	–	–	3 630 310	92 B	1 h 28 m
✓		21 600	565	21 035	–	328 175	92 B	39 m 20 s
✓	✓	21 600	565	21 035	513	322 019	92 B	37 m 40 s
✓	✓ (Rev.)	21 600	565	21 035	19 955	188 490	92 B	7 m 34 s

Table 5.2: Results for the experiments using regular TCP

We performed four different experiments on an Intel Core i7 920 2.6GHz with 4GB of RAM running Ubuntu Linux. We used Spin version 5.2.4 (running on a single core) and ns-2 version 2.33. Table 5.2 shows some measures taken from each of the four experiments. In each one we experimented with the use of the objectives and search optimization from Formulas 5.4 and 5.5, shown in the first two columns. The fourth experiment will be explained later in this section. The next four columns show data about the number of parameter configurations that were generated (from the values in Table 5.1), how many of them were tagged as *accepted* or *rejected*, and how many were not even executed due to the search optimization, respectively. The rest of the columns show the number of Spin states generated, the size of each state, and the total time of analysis, respectively, using the counter projection.

These results show that there is a notable improvement in the experiment running time when objectives are used. In addition to tagging the desired parameter configurations according to the user intentions, setting objectives helps stopping simulations early and thus reduces the space state that needs to be explored. In this case, when objectives are used the state space is about 11 times smaller and the total analysis time is about half. Using the described search optimization reduces the number of simulations that have to be executed because their outcomes can be inferred. This can be seen in the sixth column of the third row, where 513 simulations (of the total 21600) did not have to be executed to check the satisfiability of the objective, which yields an additional 4% time improvement. It is worth noting, however, that the results themselves (fourth and fifth columns) do not change when using the search optimization, and that the number of parameter configurations that are generated is always the same (third column).

The use of objectives and optimization significantly improves the total time spent on the analysis. However, not all optimizations that can be applied have the same impact. For instance, we can optimize this analysis further if we generate buffer length values for the parameter configurations in reverse and apply the converse optimization: if a simulation with a given buffer length is rejected, just decreasing the buffer will result in an equally rejected simulation:

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

$$rejectSimilar: \quad bufferLength < bufferLength' \wedge \left(\bigwedge_{p \neq bufferLength} p = p' \right) \tag{5.6}$$

The results using this optimization are shown in the fourth row of Table 5.2. The total time spent in the analysis is decreased dramatically: it is almost 5 times better than the previous optimization, and over 11 times better compared with the initial case with no objectives. This is due to having many more simulations that are rejected (21035) rather than accepted (565), which gives greater opportunities for optimization if we focus on discarding rejected simulations. By generating the buffer length values in descending order, we allow this optimization to be effective by exploring the parameter configurations in the right order.

Regarding the objective of our case study, Figure 5.10(a) shows the influence of the parameters on the evaluation of the objective. The graph on the left plots the maximum TCP window size against the playout buffer length for several values of TCP segment size. As the window size increases video playback can start sooner, since the playout buffer fills more quickly. The graph on the right shows the converse relationship, plotting the TCP segment size against the playout buffer length. From that graph it can be deduced that using larger segments means less video has to be buffered before playback can start. It follows from the results that no parameter configuration with a playout buffer of less than 10 seconds satisfies the objective. This is due to a combination of the video length and the disconnections during the transmission, i.e. the sum of the duration of the disconnections suffered during the video transmission is such that, combined with the slow start phase of TCP, it is almost impossible to overcome this gap with less than 10 seconds of margin.

Now we analyze the influence of the FreezeTCP protocol in this scenario, which should provide better performance in the presence of predictable network disconnections. FreezeTCP depends on the ability of the mobile device to predict impending network disconnections. We simulate the disconnection notifications that should come from the physical layer and automatically assign them a given amount of time before the disconnection happens. The time when the freeze warning must be sent is also a sensible parameter of FreezeTCP, and as such it is included as a parameter to optimize it.

Figure 5.10(b) shows the influence of the freeze warning on the performance of this experiment. Freeze warnings are measured in RTTs (round-trip times) and, according to [69], ideally they should be sent one RTT before the disconnection. However, as can be seen in the figure, this also depends on the duration of the link downtime periods. A lower downtime means that the warning can be sent some time before the disconnection really happens, putting less pressure on the prediction algorithm. The higher the downtime, the less important it is to predict it precisely, as the duration of the disconnection is considerably larger than the idle time wasted freezing the connection sooner. Comparing this graph with the ones in Figure 5.10(a), it can be seen that the freeze mechanism

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

Link configuration		TCP configuration	
Bandwidth	Delay	Segment size	Max. window
288 Kbps	150 ms	204 bytes	10 KB
320 Kbps	100 ms	204 bytes	8 KB
384 Kbps	125 ms	255 bytes	10 KB

Table 5.3: Parameter selection for some link configurations

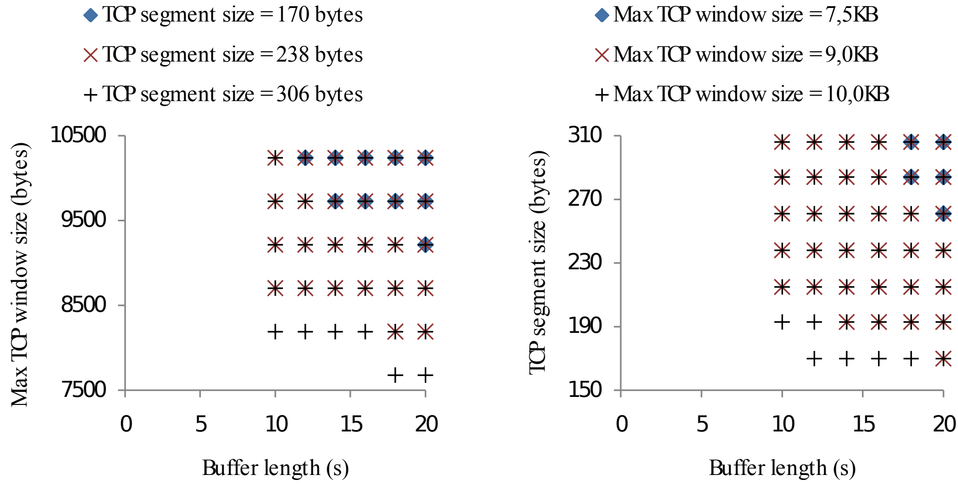
allows the use of a smaller playout buffer, as the effect of the disconnections is partially mitigated.

In addition, we prepared a series of scenarios to measure the influence of the precision of variables in the analysis, i.e., how the number of values for a variable can affect analysis time. We start with the FreezeTCP scenario, set fixed values from some of the variables, try 10 values for the freeze warning variable and 4 to 20 values for the buffer length variables. Figure 5.11(a) compares the number of simulations that are accepted or rejected by the objectives against the number of simulations that are not run due to the optimizations. Depending on the optimization used, only one of these two groups of simulations was not run. Although the number of rejected simulations almost doubles the number of accepted simulations, the difference in analysis time shown in Figure 5.11(b) by using optimizations is not as dramatic as in Table 5.2, where the ratio between those two sets is much bigger. Nevertheless, time of analysis is clearly improved by using objectives and optimizations, and accepted parameter configurations are still found.

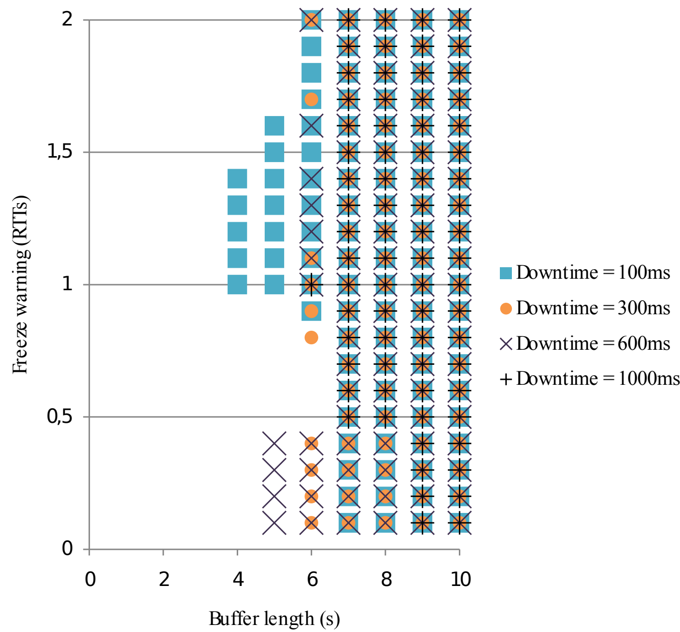
Optimization

Finally, we apply the results from the simulations performed to adapt the video transmission to changes in the link during the transmission. By analyzing these results we can determine which parameters should be used in a particular scenario with known values of link bandwidth and delay. However, it is likely that these parameters will change during the course of the video download, which may cause our no-rebuffering objective to fail because the initial selection of protocol parameters is not appropriate for the new situation. To validate our approach, we prepared a scenario where the bandwidth and delay of the link change after each disconnection, e.g. after moving to a new cell with slightly different conditions. After each reconnection, the TCP entity on the server side reconfigures itself to adapt to these changes, changing its TCP segment size and maximum window size values. We selected the combinations of parameters that perform the best in each scenario, based on whether the objective was met and the time to complete the video download, some of which are shown in Table 5.3. We assume fixed values for the rest of the variables.

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA



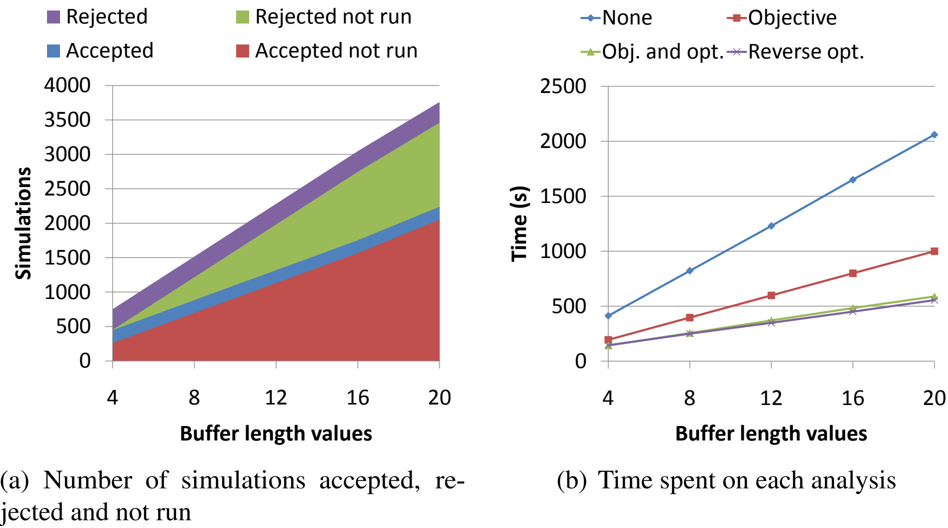
(a) Influence of maximum TCP window size and TCP segment size in the first experiment



(b) Influence of freeze warning in the second experiment

Figure 5.10: Influence of parameters in the experiments

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA



(a) Number of simulations accepted, rejected and not run

(b) Time spent on each analysis

Figure 5.11: Influence of the precision of variables

We compared the adaptive version of the FreezeTCP protocol with three non-adaptive versions. The link configuration of the scenario changed after each disconnection, using the parameters from Table 5.3. The adaptive version used the appropriate configuration after each reconnection, while the non-adaptive versions were fixed on one of the configurations from Table 5.3. To highlight the importance of using an appropriate configuration, we used a higher downtime than usual of 3 seconds per disconnection.

The adaptive version met the objective we set and outperformed the other simulations, which had to stop several times for rebuffering during playback. Figure 5.12 shows the instantaneous throughput of the adaptive version as compared to the three non-adaptive versions using fixed parameters, highlighting the playback interruptions that the latter experienced.

5.4 Case study: E-model extension

The second case study deals with the evaluation of VoIP quality of experience (QoE) [105]. We developed an extension of the E-model [10][1], based on the improvements proposed by Clark [39]. In this section, we perform a validation analysis of this new E-model.

5.4.1 E-model for VoIP QoE evaluation

The E-model [10][1] is a computational model for objectively estimating end-to-end voice call quality in packet-switched networks. This model takes into account several parameters that may affect the quality, such as delay or packet loss. The main result of the

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

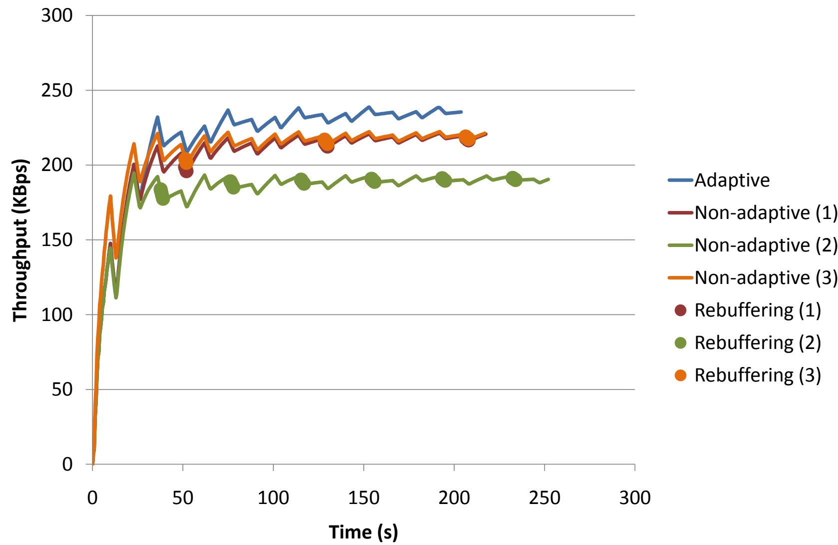


Figure 5.12: Comparison of adaptive vs. non-adaptive of video download throughput

E-model is a rating, called the R factor, which can then be transformed to obtain a MOS (mean opinion score) value. The MOS is a subjective quality score usually associated with human evaluation, which the E-model tries to approximate objectively.

The formula for computing the R factor with the E-model is as follows:

$$R = R_o - I_s - I_d - I_e + A \tag{5.7}$$

In this formula, Ro is the basic signal-to-noise ratio, I_d represents the effects of delay, I_e the effects of the codec and other "equipment", I_s represents other simultaneous impairments, and A is the advantage factor, i.e. the allowance users make due to the convenience of using a given technology (e.g. a cellphone). Assuming default values for some of the parameters (per [10]), the formula can be simplified to:

$$R = 94 - I_d - I_e \tag{5.8}$$

To obtain the MOS score from the R factor, the following formula from [10] (with a correction from [35]) can be applied:

$$MOS = \begin{cases} 1 & R < 6.5 \\ 1 + 0.035R + 7 \cdot 10^{-6} R(R-60)(100-R) & R \in [6.5, 100] \\ 2.5 & R > 100 \end{cases} \tag{5.9}$$

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

Clark's E Model

Clark's E-model [39][2] takes into account some of the time varying impairments that can affect the perceived quality of a call. In particular, "bursty" packet loss is identified as a source of significant call quality degradation. Isolated packet losses during periods without packet losses yield better quality perception, while a burst of lost packets is perceived as being of worse quality, even if the average packet loss of the call is the same.

Another consideration of this model is that quality perception changes smoothly during the call. The "instantaneous quality" of a single time instant of the call can be computed, but the "perceived quality" that the user would report at that instant depends on the recent history of the call. In addition, when the instantaneous call quality is worse, i.e. during a burst, the perceived quality decays faster than it recovers when the instantaneous quality is good. Instantaneous call quality can be computed from the packet loss ratio, using a function that depends on the VoIP codec used [1][2].

For the purpose of modeling bursty packet loss and computing call quality, Clark uses a 4-state call model, which can be separated into two groups: burst and gap. A burst indicates a period of high packet loss, while a gap indicates a period of low packet loss between bursts. More formally [10], a burst is the longest sequence that a) starts with a lost packet, b) does not contain any occurrence of g_{min} or more packets received consecutively, and c) ends with a packet loss. A gap is defined as a) the period between the start of the session and the last received packet before the first lost packet, b) the period between the end of the last burst and the end of the session, or c) the period between two bursts. Thus, a lost packet within a gap must be preceded and succeeded by at least g_{min} packets. g_{min} is the parameter that defines the minimum gap size, and is typically set to 16.

Figure 5.13 shows Clark's 4-state Markov model, composed of the product of call state (burst or gap) by packet event (loss or reception). Transitions due to packet losses are marked with a dotted line. The transitions on the figure are also labeled with the probabilities p_{xy} of transitioning from state x to state y.

These probabilities can be computed by counting state transitions in variables c_{xy}. The event-driven algorithm from [10] (in turn taken from [2]) shown in Listing 5.3 computes these counters. Note that some counters can be derived from other counters, so they have not been included in the algorithm, e.g. `c31 = c13` and `c32 = c23`. For the sake of simplicity, we consider that discarded packets are lost packets. In this algorithm, `pkt` counts the number of packets received since the last lost packet and `lost` counts the number of lost packets in the current burst. `packet_lost` is a boolean variable which is true when the event was triggered by a lost packet.

The average quality level $I_e(av)$ of a call can be calculated as follows. Let I_1 be the quality level at the change from a burst with quality I_{eb} to a gap with quality I_{eg}. Let I_2 be the quality level at the change from a gap with quality I_{eg} to a burst with quality I_{eb}. Let b be duration in packets of the current or last burst, and g the duration in packets of

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

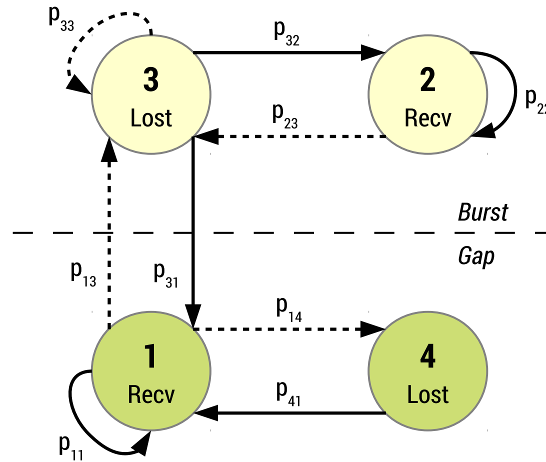


Figure 5.13: 4-state Markov model of bursty packet loss

the current or last gap. I_1 and I_2 are defined as follows:

$$I_1 = I_{eb} - (I_{eb} - I_2)e^{-b/t_1} \tag{5.10}$$

$$I_2 = I_{eg} + (I_1 - I_{eg})e^{-g/t_2} \tag{5.11}$$

Combining and integrating equations 5.10 and 5.11 to give a time average gives:

$$\begin{aligned} I_e(av) = & \frac{bI_{eb} - t_1(I_{eb} - I_2)(1 - e^{-b/t_1})}{b+g} \\ + & \frac{gI_{eg} + t_2(I_1 - I_{eg})(1 - e^{-g/t_2})}{b+g} \end{aligned} \tag{5.12}$$

Clark also proposes a simple model for the delay impairment factor, I_d. This model is based on the observation that a delay below 175ms has little impact, while the perceived quality decreases significantly from that point. Given an average of the delay over a period of time, the I_d factor in that period is computed as follows:

$$I_d = \begin{cases} 4 * \text{delay} & \text{delay} < 175\,\text{ms} \\ 4 + (\text{delay} - 175)/9 & \text{delay} \geq 175\,\text{ms} \end{cases} \tag{5.13}$$

Clark's E-model, or parts of it, have been used in some methodologies for evaluating VoIP call quality, including Clark's own VQmon tool[1]. In [91] the way of users reacting different to bursts and gaps, and the effects of recency, are incorporated as part of a

[1] http://www.telchemy.com/vqmon.php

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

```
if (!packet_lost) {
    pkt++;
}
else {
    if (ptk >= gmin) {    // Gap
        if (lost == 1) {
            // Isolated loss in a gap
            c14++;
        }
        else {
            // Coming from a burst
            c13++;
        }
        lost = 1;
    }
    else {                // Burst
        lost++;
        if (pkt == 0) {
            // Consecutive packet losses
            c33++;
        }
        else {
            // Loss after pkt packets
            c23++;
            c22 += ptk - 1;
        }
    }
    pkt = 0;
}
```

Listing 5.3: Clark's algorithm for counting state transitions of bursty packet loss model

methodology to evaluate VoIP calls over Internet backbone networks. The authors found a trade-off between loss and delay: losses can be decreased with a larger playout buffer, which leads to higher delays. Also, it was found that paths with low delay and low delay variability lead to better MOS scores. Under-provisioning and heavy traffic were identified as some of the possible causes of problems with the delay.

Carvalho et al [35] implemented this E-model and used it to evaluate the characteristics of Internet backbones in Brazil. The experiments showed differences in the MOS scores for upstream and downstream calls, attributed to the asymmetry of the connections. Load experiments were also performed, resulting in significant QoE degradation between as little as 5 and 10 simultaneous calls in one of the directions.

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

5.4.2 On-the-fly E-model computation

The algorithm given in [39] does not accurately count all transitions [35]. The first packet of a burst is always incorrectly classified as an isolated loss in a gap. Counter `c14` is always incremented with the first packet loss in a gap, even if that packet loss is really the first one in a gap (line 8 in Listing 5.3).

Also, this algorithm is not able to compute the MOS while the call is in progress. Having an on-the-fly QoE evaluation would be useful in analyses using our OptySim framework, as we have discussed in this and previous chapters. This would allow to stop simulations early when using objectives relative to the QoE evaluation.

However, the algorithm discussed so far is not able to give a good enough account of the call state while it is in progress. Gaps can only be detected after `gmin` packets are received, so it is not possible to tell whether a series of packet receptions is part of an ongoing burst or a new gap before receiving `gmin` packets. During these periods there are two options: a) keep reporting the last known state but with updated (and possibly wrong) counters, or b) report the last known state and the last precisely known counters. If used on-the-fly, the algorithm in Figure 5.3 would report only known state transitions (with one exception, as we have established), but the counters are only updated after a packet loss. During long gap periods without isolated losses, this algorithm would not update the call state and counters.

We have developed a new algorithm that solves these shortcomings. This algorithm deals with uncertainty, i.e. instants where the current state of the call cannot be accurately determined, by delaying transition counting until it can be resolved. At the same time, counters are updated as soon as possible, giving better real time results than those that the previous algorithm could give.

Figure 5.14 shows a simplified state machine of this new algorithm. Note this is not the model of burst packet loss, which is still the same as in Figure 5.13. This algorithm reacts to both packet reception (*recv*) and loss (*lost*) events. States with square corners are certain states, while uncertain states are represented with rounded corners.

While in an uncertain state, counters are not updated and temporary counters are used. Upon reaching a certain state, counters are updated accordingly from these temporary counters. For instance, while in the "Gap recovery" state the call may return to the "Gap" state or may go to the "Burst" state. In the first case, counters `c14` and `c41` should be updated to reflect an isolated loss, together with counter `c11`. In the second case, counter `c13` should be incremented to account for the transition from gap to burst, and counters `c32` and `c23` should be incremented to account for a short gap of received packets, which will be added to `c22`.

Appendix C contains the complete algorithm for detecting and counting the transitions between states. For instance, an isolated loss would be treated as follows. After a series of received packets, a single `lost` event is triggered. The state is changed from `GAP` to `FIRST_LOSS`, and the current time is saved in `startUncert` as the possible start of a burst.

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

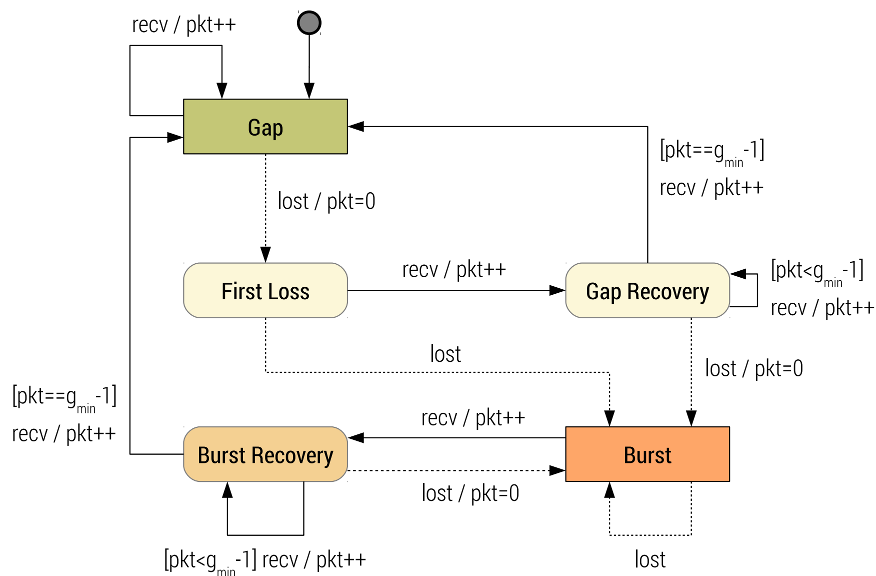


Figure 5.14: Simplified state machine of the improved algorithm for burst modeling

The next packet is received, with `recv` making the transition to `GAP_RECOVERY`. Note that we cannot tell if this is an isolated loss yet. Uncertainty is managed in this state using the `endUncert` time mark and the `pkt` received packet counter. The latter is increased, but is not added to any of the counters. After more than `gmin` packets have been received in a row, `recv` changes the current state to `GAP` and updates all pending counters. It is at this time that a more up to date value for the MOS score can be computed, since the model is back in a certain state.

5.4.3 Implementation in ns-2

We have implemented three E-models in ns-2: a simple version based on the original definition of the E-model, Clark's improvements on the original (both discussed in Section 5.4.1) and our own improvements on Clark's version (Section 5.4.2).

These E-models were implemented on top of ns2Voip++ [25], an existing VoIP extension for ns-2. ns2Voip++ is provided as a patch for ns-2, mixed with ns2measure [37], a framework for data collection and statistical analysis. The first step was to split the patch to separate these two components, and then turn the ns2Voip++ patch into a ns-2 dynamic module. This module was modified to add our E-model classes. Since our E-model implementations require changes to the ns2Voip++ module, it could not be implemented as a separate module. In addition, new error models were developed to create several types of error bursts, which were used while evaluating our model.

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

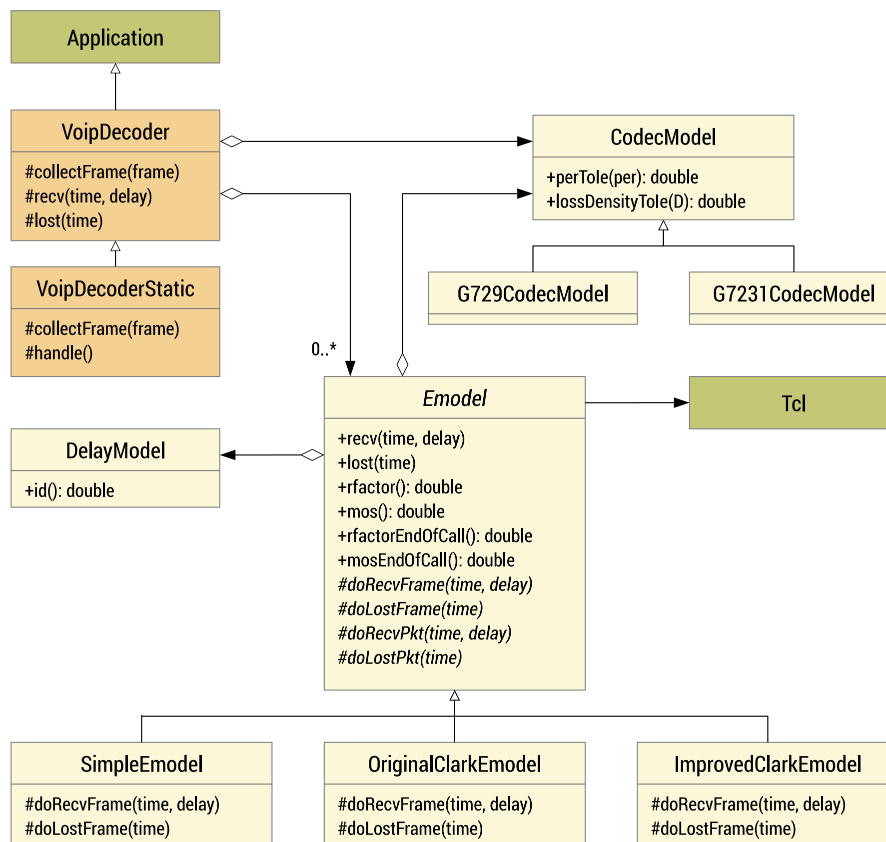


Figure 5.15: Class diagram of part of the E-model module

E-model implementation

The main classes of the module are shown in diagram 5.15. The main base class of our extension is `Emodel`, an abstract class that serves as the basis for different implementations of the E-model. For instance, `Emodel` declares pure virtual `doRecvFrame`, `doLostFrame`, `doRecvPkt` and `doLostPkt` methods, to be used as frame and packet event hooks, and the pure virtual `rfactor`, which computes and returns the current value of the R factor. In addition, as described in Section 5.4.1, at the end of a call, the time that has elapsed since the last significant error burst affects the perception of the overall call quality. Subclasses that support this adjustment can override the `rfactorEndOfCall` method to provide a proper end-of-call value.

The module provides three different `Emodel` implementations. `SimpleEmodel` simply takes into account the number of packets received and lost, ignoring bursts and gaps. `OriginalClarkEmodel` implements the original algorithm as described in Section 5.4.1. On the other hand, `ImprovedClarkEmodel` implements our improved algorithm as described in Section 5.4.2.

Publicaciones y Divulgación Científica
uma UNIVERSIDAD DE MÁLAGA

The ns2Voip++ module was modified to add the required hooks to handle packet events in the receiver. VoIP packet buffering and playback is handled by `VoipDecoder` and its subclasses, which implement different algorithms for this task. For our module, we only modified the `VoipDecoderStatic` class, adding the corresponding triggers to notify the `Emodel` instance being used.

The simple delay model described in Section 5.4.1 was implemented in `DelayModel`. The base `Emodel` class reports the instantaneous delay values to a instance of this class so the delay impairment factor (I_d) is always available to the E-model implementations when they need it.

Computing Ie from PER

The equipment impairment factor (I_e) depends on the characteristics of the codec used. Some codecs provide better inherent quality or are more resilient to errors than others. ITU-T Rec G.108 [1] provides provisional values of I_e for codecs G.729-A and G.723.1 under conditions of packet loss. However, these values are provided only at some discrete points from 0% to 16% packet loss. On the other hand, ETSI TS 101 329-5 [2] provides a formula which, given the loss density of a period of time, yields very similar I_e values. Unfortunately, the formula has been defined for codec G.732.1 but key parameters are missing for G.729-A.

We decided to implement both methods of computing the I_e from the packet error rate (PER) in the `CodecModel` classes. The base `Emodel` class can be configured so that the method used can be set as a per-instance preference. When using the ITU-T Rec G.108 table, intermediate values are interpolated from the given discrete points. For PER values greater than 16%, I_e values are extrapolated.

Error models

We have included several new error models to test our E-model module, shown in Figure 5.16. The abstract base class `PerErrorModel` provides some common functionality for the error models, such as computing the resulting PER. The first error model, `VoipErrorModel` wraps another `ErrorModel`. ns2Voip++ includes key information for the decoder in the first packet of a talkspurt, so we worked around this limitation by preventing these packets from being lost with `VoipErrorModel`. `BurstErrorModel` is a two-state error model with gap and burst state, of fixed length (measured in packets), which are alternated. Each state relies on another `ErrorModel` to decide whether a packet should be corrupted.

As mentioned in the previous section, when using the discrete points method an extrapolation is required for PER values greater than 16%. Using random variables it is possible to inadvertently create bursts with higher error rates. To avoid this, we implemented `LimitedBurstErrorModel`, another error model which creates predictable

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

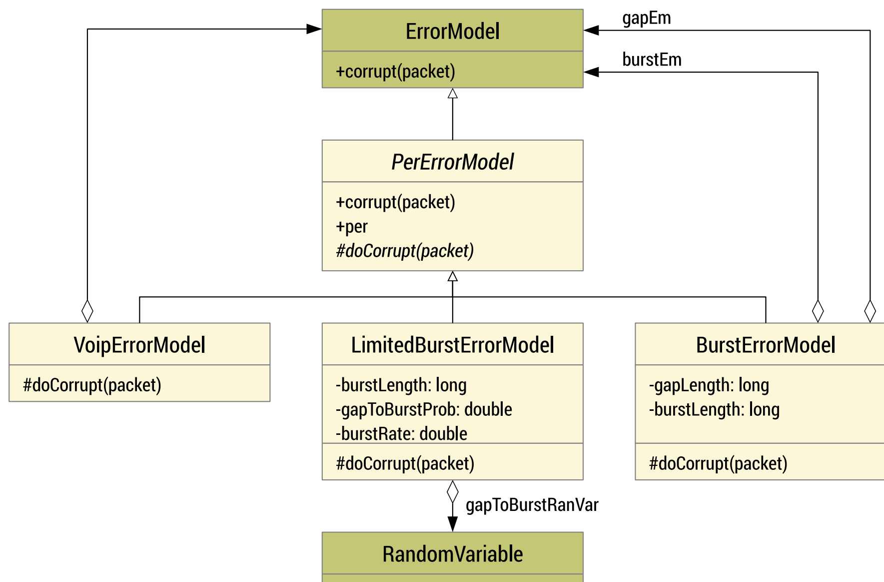


Figure 5.16: Class diagram of the new error models

bursts with the given PER by corrupting packets at regular intervals. The occurrence of the bursts themselves is governed by a random variable.

5.4.4 Validation analysis

We set out to validate our E-model extension. In particular, as our extension is a refinement of Clark's, we were interested in its conformance to the results produced by the latter extension.

Given the differences between the E-model extensions we did not expect the same results by comparing the two E-models. Although both extensions are based on the same principle and computations, ours reacts promptly to changes in the state of the call, while the original performs a single average computation of all previous states. Thus, our extension is more sensitive to the particular error patterns in a call (more details can be found in [105]). However, both E-models still approximate a subjective measure. Therefore, we argue that, in this case, we can establish conformance with a given E-model implementation as obtaining similar MOS scores when analyzing the same calls.

The validation scenario and objective are defined as follows. Given a series of N independent, parallel VoIP calls, each one of them measured by two independent E-model instances, namely Clark's extension and our own extension, we define the validation objective of a single simulation as follows. Informally, we require that the MOS scores reported by at least M pairs of E-models are within R distance, measured in absolute

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

terms, for at least T seconds. Note that this is not a condition expressed in the E-model literature, but rather one we imposed on our implementation. Formally, we define the objective as:

$$
\begin{aligned}
\mathit{ltlAccept}: &\quad \Diamond(\mathit{inside}* \wedge (\mathit{inside}*\ U\ \mathit{timeUp})) \\
\mathit{def}: &\quad \mathit{timeUp} \;:=\; \mathit{time} - \mathit{startOf} >= T \\
\mathit{def}: &\quad \mathit{inside}* \;:=\; \mathit{in}_i(5) + \mathit{in}_i(4) + \mathit{in}_i(3) + \mathit{in}_i(2) + \mathit{in}_i(1) \geq M \\
\mathit{def}: &\quad \mathit{in}_i(i) \;:=\; \begin{cases} 1 & \mathit{if}\ \mathit{in}(\mathit{mos}[i][1][\mathit{mean}], \mathit{mos}[i][2][\mathit{mean}]) \\ 0 & \mathit{else} \end{cases} \\
\mathit{def}: &\quad \mathit{in}(\mathit{val}, \mathit{ref}) \;:=\; \mathit{val} \geq (\mathit{ref} - R)) \wedge (\mathit{val} \leq (\mathit{ref} + R))
\end{aligned}
\tag{5.14}
$$

inside* evaluates to true if the MOS scores of at least M E-model pairs are within R distance, and is defined in terms of the measured MOS scores of each flow. These scores are contained in separate variables, but for clarity we have represented them here as the four-dimensional array *mos*, e.g. $\mathit{mos}[i][2][\mathit{mean}]$ is the mean of E-model instance number 2 of the i-th VoIP node. The definition of *timeUp* requires the use of a variable, *startOf*, to mark the period in which inside* holds true, which cannot be done directly in the LTL formula. Instead, we provide this variable manually from the model. The value of this variable is computed in the model before each state is sent to the analyzer.

To perform the analysis for this objective, we declare a projection with the required model variables. On the one hand, we need the simulation time, which is included by default. On the other hand, we need the current value, average and standard deviation for the MOS values obtained for each E-model instance, properly named to distinguish to which VoIP flow and type they belong. Finally, the artificial variable *startOf* is also included.

In contrast with the case study of Section 5.3, here we do not project states when the relevant variables are updated, but rather at regular intervals. This is the time limit for the folded projection mentioned in Section 4.2.3, and is generally useful when dealing with continuous variables, such as the MOS and related statistics. Another possibility would have been to discretize the values of these variables so that only changes above a certain threshold are projected into the abstract trace.

To perform an exhaustive validation, we set a parameterized scenario with different error models and codecs. The scenario includes five concurrent VoIP flows, with $M = 3$, $R = 0.1$, and $T = 30s$. We measure the MOS scores of each E-model instance at regular intervals. Given the objective declared in Equation 5.14 and the parameterized scenario, OptySim analyzes each scenario instance in isolation, accepting or rejecting our E-model in each one.

The previous validation objective defines "local" validation, i.e. the fitness of an E-model implementation for a very specific scenario. We extend these validation results

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

Error Mode		Codec		VoIP frames/packet	
Random	*100 %*	G.723.1	*100 %*	1	*100 %*
				2	*100 %*
		G.729A	*100 %*	1	*100 %*
				2	*100 %*
Burst	*89.6 %*	G.723.1	*88.3 %*	1	*90.9 %*
				2	*85.6 %*
		G.729A	*91 %*	1	*91.9 %*
				2	*90 %*
Limited	*78.5 %*	G.723.1	*75.2 %*	1	*80 %*
				2	*70.3 %*
		G.729A	*81.8 %*	1	*86.5 %*
				2	*77 %*

Table 5.4: Validation results for our E-model extension

over a range of conceived scenarios to come up with "global" validation results, measured as the percentage of validated scenarios. Since the scenario parameters induce some natural groups, i.e. according to the error model used, we can also compile the validation results in these groups.

After the tool has completed the analysis, we collect the local results to compute the global results. A breakdown of these is shown in Table 5.4, grouped by error model, codec and VoIP frames per packet. From these results we can conclude that our extension is a perfect fit for the random case. However, when using bursty error models ("Burst" and "Limited"), validation scores are weaker. This is due to the fact that our extension takes each burst and gap into account separately, thus being more sensible to the particular error patterns.

Another interesting result is that fitting two VoIP frames in each packet yields noticeably worse results. Again, this can be attributed to our extension working with a finer grain. A packet with two frames lost during a gap introduces a small burst with an error rate of 1.0. This induces a small but noticeable degradation in the instantaneous call quality estimation.

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

Chapter 6

Inverse modeling of network KPIs

Network simulations are a cost effective solution for the development and planning of network deployments. They enable quicker, cheaper analysis and iteration than what would be possible with field tests. However, field tests are still the only way to capture real environmental conditions that are usually not taken into account in simulations. Investing in more realistic network models would improve the quality of analysis yielded by network simulations and reduce the dependency on field tests.

In the field of cellular network performance evaluation there are factors, such as latency, jitter or dropped packets, which have significant influence in performance. These factors, called key performance indicators (KPIs) [86], may be the result of several environmental conditions. Obtaining self-contained models of these KPIs, with the same characteristics observed in field tests, would enable more realistic and useful simulations.

In this chapter we show how OptySim can be applied to the inverse modeling of network KPIs, i.e. obtain a model that fits the collected data about a KPI. In particular, we tune a parameterized jitter model for mobile VoIP using data collected from field tests. This data enables acquiring a deep understanding of mobility issues from the application level point of view and obtaining realistic results regarding the quality of service perceived by final users. For the data collection phase we used the SymPA [53] profiling tool, while network simulations for the analysis phase are carried out using the ns-2 network simulator.

6.1 Key performance indicators

Key performance indicators (or KPIs) are computed measures that are related to the performance of a system [86]. The concept of key performance indicators is not exclusive to communications networks and can be found in several other areas, such as economy, and does not necessarily deal with technical details. Usually, KPIs are used as an optimization target, e.g. for business or network planning. KPIs are computed from raw

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

input data, such as counters and timers, or from other KPIs. There is no restriction on the complexity of said computation, as long as it is not raw data.

3GPP has defined several KPIs for their radio access technologies, such as GSM and UMTS [87]. These may be based on measures from different parts of the network, such as the core network (either circuit- or packet-switched), the IP multimedia subsystem (IMS) or the base stations. Other KPIs for these radio access technologies can be found in the literature [86]. In the case study which can be found later in this chapter we will work with a simple KPI: jitter. Jitter can be defined informally as the difference between the expected and the actual packet arrival times.

In the first stages of performance evaluation, network simulations and other means of off-line analysis are vital. In the field of cellular network performance evaluation, numerous analytical models have been proposed for modeling some of the factors that impact performance [98][50]. These models provide theoretical results based on assumptions, usually with the objective of reducing associated computational costs. However, sometimes these assumptions lead to models that are too simplistic or that do not properly capture some particular circumstances. Approaches that produce more realistic models are thus welcome [71], as they will improve the quality of the analysis than can be performed without resorting to costly field tests.

Thus, our objective for this application of our framework is obtaining realistic KPI models to be used in simulations, which match field test observations, and providing a methodology for doing so.

6.2 Inverse modeling methodology

This section describes our methodology for the inverse modeling of KPIs for performing more realistic simulations, using our objective-driven analysis framework.

The process starts with a parameterized KPI model that has to be configured appropriately to match the observations of the field tests. This model declares a parameter space that has to be explored in order to find the best fitting configurations. The fitness of a given configuration can be evaluated by comparing a series of simulations with real traces from the field tests. However, the number of possible configurations may be too large to be simulated exhaustively. Thus, we introduce OptySim with two purposes: automatically finding the best configurations by declaring a fitness objective, and reducing analysis time by terminating unproductive simulations early.

For the field tests we use SymPA [53], a profiling tool for mobile phones. Using SymPA we can extract relevant data and metrics regarding a KPI from field tests. For modeling and simulating KPIs, we use the ns-2 network simulator. This methodology is independent on the particular simulator (or profiling tool) used. However, we already have the proper integration between ns-2 and our framework, as described in Section 5.2.

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

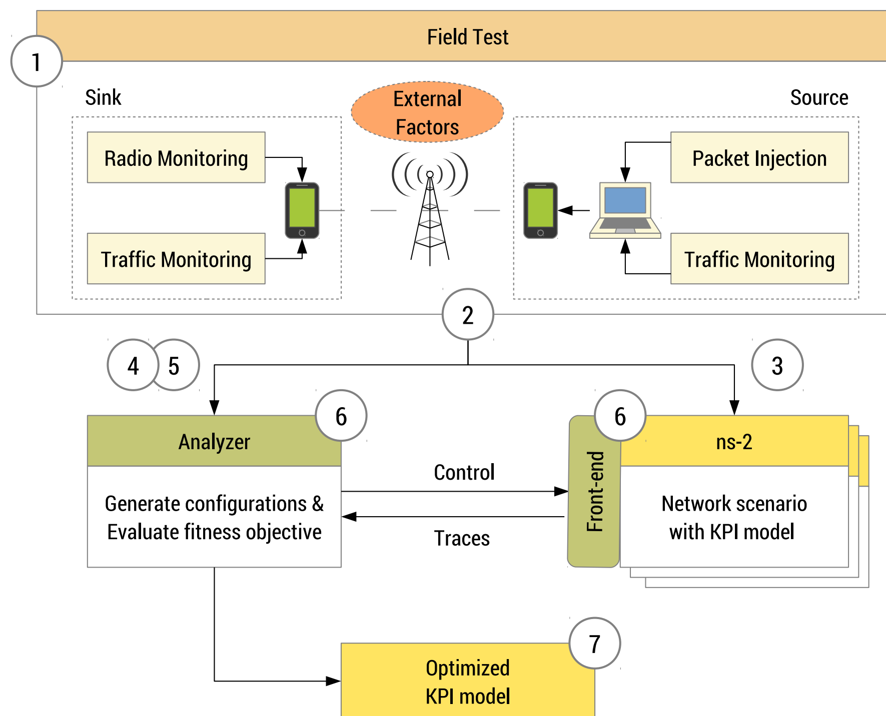


Figure 6.1: Overview of methodology for inverse modeling of KPIs.

Figure 6.1 shows an overview of the methodology. Given a parameterized KPI model, the steps for tuning it to match the characteristics observed in field tests are as follows:

1. **Field tests.** Decide the environment which the KPI should model, e.g. a mobile environment. Perform a series of field tests to capture real traffic traces of mobile devices with a profiling tool, such as SymPA. Additional data can be captured from other network endpoints if appropriate.

2. **Collect KPI data.** Gather the relevant characteristics of the KPI from the collected traces. The particular characteristics that should be collected depend on the fitness objective that will be used later in the methodology. However, it is usually a good idea to gather general statistics, such as average and standard deviation.

3. **Define ns-2 scenario.** Build a ns-2 scenario with the same setup as the field tests. using the parameterized KPI model in the appropriate place. For instance, a jitter KPI model could be placed in the mobile link of a network scenario.

4. **Declare parameter space.** Identify the parameters of the KPI model and their valid ranges. Declare them in the analysis configuration file (see Section 3.1.1).

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

5. **Declare fitness objective.** Declare the fitness objective that the KPI model must meet, with regards to the data gathered in step 2. This objective is also declared in the analysis configuration file. The form of this objective (or objectives) is up to the user, which can use any of the properties that OptySim understands, such as LTL formulas or asserts. The case study in Section 6.3 includes an objective written as an LTL formula, which can be reused for other purposes. This objective checks that some simulation parameters stay within certain bounds for a given period of time.

6. **Analysis.** The analysis is carried out normally, as described in Chapter 3. Spin generates parameter configurations for the model, which are then simulated in ns-2. In parallel, Spin evaluates the objective on the execution traces generated during these simulations. Each simulation is tagged accordingly.

7. **KPI optimization.** Tuned parameters for the KPI model, i.e. the parameters that satisfied the fitness objective, are returned as a result of the analysis. The model can then be used to perform realistic simulations with regards to the KPI in the circumstances recorded in the field tests.

6.2.1 SymPA

SymPA (Symbian Protocol Analyzer) [53][56] is a network monitoring tool targeted at Symbian smartphones. Its key feature is the ability to capture IP traffic and radio access technology measurements on commercial mobile devices. The correlation of the collected data allows determining the source of connectivity problems, such as packet losses or higher latency, and detecting strategic parameters in the mobile communication optimization process. SymPA has recently been ported to Android smartphones, under the name of TestelDroid [24].

SymPA has been used to carry out extensive field tests in real scenarios to characterize the behavior of video streaming services over cellular networks [54]. The methodology used was based on using off-the-shelf smartphones to obtain more realistic results which are beyond the scope of the measurements provided by the internal probes used by mobile operators to monitor the performance of the core network.

6.3 Case study: jitter model

Jitter is an important factor that impacts the performance of real time applications, such as VoIP or video streaming, and as such obtaining a realistic jitter model for simulations would be of great value. In this section we describe how we applied the methodology described in the previous section to obtain a realistic jitter model. We also show a naive

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

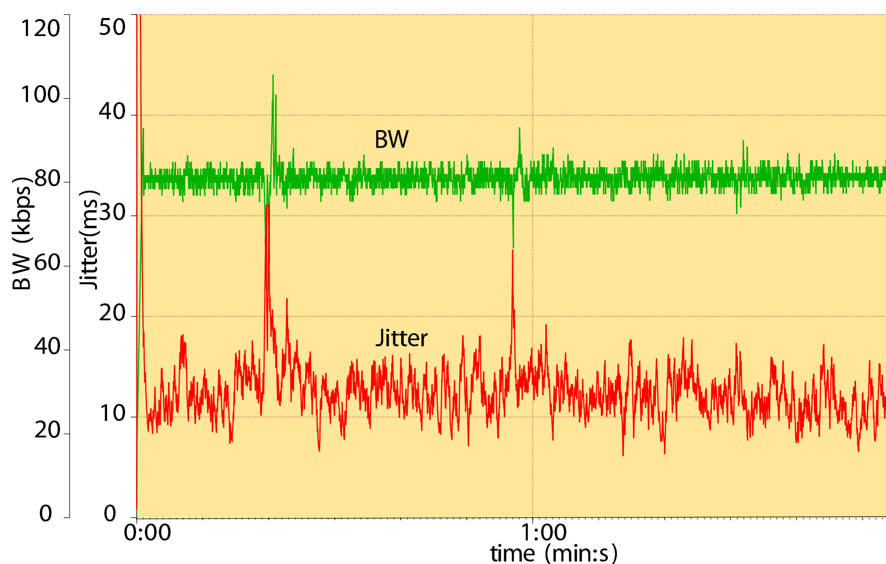


Figure 6.2: Jitter and bandwidth captured during one of the HSDPA field tests.

approximation for jitter modeling, in which jitter traces from fields tests are replayed during a simulation.

We use parameterized probability distributions to model the jitter observed in the field traces. In particular, we try to tune these parameters to approximate the average and standard deviation of the jitter. Different combinations of these parameters were automatically generated and simulated in order to select the configurations that more closely matched the measured jitter. Although we present the results for ns-2, the jitter models obtained could be used in any network simulator that supports custom delay models for links.

6.3.1 Field test scenario

The field test scenario consisted of a RTP (Real Time Protocol) traffic generator running on a desktop computer using a mobile phone as a modem. At the other communication endpoint there was a mobile device capturing the incoming RTP packet flow injected at the source. Traffic capture was performed directly at the IP level in the mobile device using SymPA [53]. The tests were performed using two different radio access technologies: UMTS (Universal Mobile Telecommunications System) and HSDPA (High-Speed Downlink Packet Access. A sample of the results obtained from the analysis of traffic captures during field tests is shown in Figure 6.2.

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

6.3.2 Replaying jitter traces in simulations

A simplified version of the field test scenario was modeled in ns-2, comprising of two UDP nodes connected with a duplex link. One of the nodes had a constant bit rate traffic generator acting as a RTP traffic generator, while the other end was a custom UDP sink used to compute jitter and other statistics. The link between the two nodes is an abstraction of the network path (both wired and wireless) between the desktop computer and the mobile device. However, since we only measure both ends of the path, it is adequate to model the measured characteristics of the connection. ns-2 provides a range of error models to simulate packet loss in a link. Although these error models can also delay some packets, the delay that is applied is always constant and thus a poor fit to model jitter. Therefore, custom error models were developed in C++ to inject a variable delay in the link.

Our first approximation towards using field test data for performing more realistic simulations was to use the recorded traces to reproduce the same jitter and packet loss. Thanks to the traces captured at the RTP sink, we had both the sequence numbers of the received packets and the inter-arrival time of each pair of consecutive packets. From the sequence numbers we were able to extract the loss bursts that could be provided as input for `ErrorModel/Trace`, a ns-2 built-in error model class based on a special trace file. Following the same principle, we developed the `ErrorModel/JitterTrace` class, which applied an additional delay to recreate the inter arrival time read from the traces. This error model was inserted after the packet queue associated with a link. Therefore, another error model could still be hooked before the queue in order to simulate packet errors and losses.

The field tests were recreated in the simulator using the trace data as input for the error models, and the same configuration in the scenario. Figure 6.3 shows the jitter measured from a GPRS test with 500 B packets, plotted against the jitter from the corresponding simulation. The profile of both graphs is very similar, but not exactly the same. This can be attributed to slight mismatches between the simulator's exact scheduler and the real traffic generator, which did not send packets at exactly the same interval, and differences in the precision of the performed mathematical operations, among others. But above all, this approach for using field test data is too rigid, depending on the parameters set for the traffic source, which would not translate well to other scenarios.

6.3.3 Inverse modeling

To overcome the limitations of the simple jitter model introduced above, we developed a parameterized jitter model which will be configured using the methodology described in Section 6.2. For this task, we will also build a ns-2 scenario that mimics one of the field tests, and we will take advantage of the data collected from the field tests, extracting relevant statistics which we will use to tune our model.

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

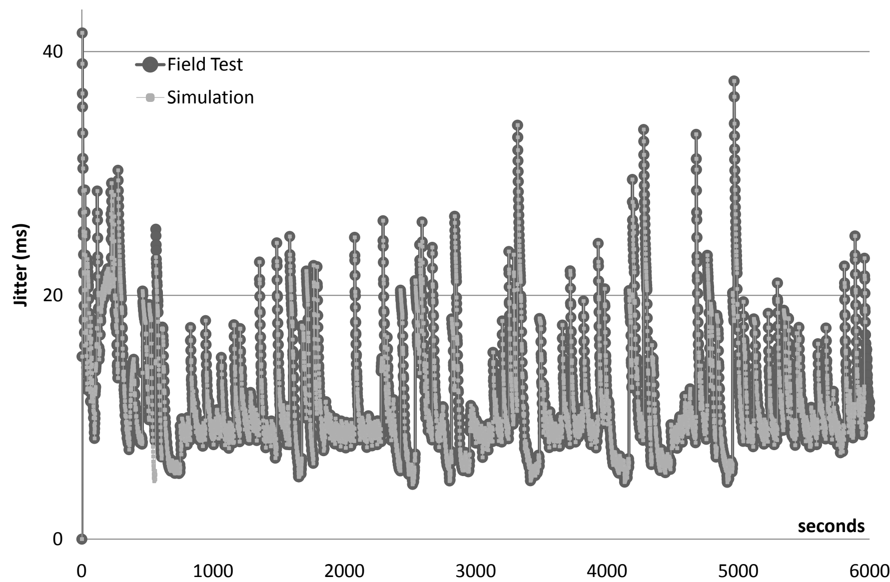


Figure 6.3: Jitter from field test data vs simulation with replay

The jitter model, `ErrorModel/Jitter`, was implemented as a ns-2 error model in C++. As with the previous `ErrorModel/JitterTrace`, instances of this class are meant to be inserted after the link packet queue. This class uses an instance of `RandomVariable` (already provided by the base `ErrorModel` class) to delay a packet a random amount of time. ns-2 provides several implementations of `RandomVariable`, each modeling a different probability distribution with their respective parameters. These are the parameters which we are interested in tuning for obtaining a realistic jitter model, i.e. the parameter space to be explored.

On the other hand, the target of the optimization objective is the generated jitter using each model. More specifically, certain jitter statistics that can be collected from the simulations will be compared against the same statistics gathered from the field tests. In this case study, the execution trace to be analyzed will be composed of these statistics, measured at different points in time during the simulation. To measure this statistics we developed a custom UDP agent, `Agent/UDP/My`, to be placed at the sink node.

Figure 6.4 shows an overview of the ns-2 simulation scenario built for the inverse methodology process. A regular error model is placed before the link queue, while the jitter model is placed before, in order to delay the packets as they exit the queue. The traffic source and the link have been configured with the same characteristics observed in each field test scenario (UMTS or HSDPA). Additional Tcl code was added to poll periodically the custom UDP agent at the sink and send the corresponding statistics to the analyzer.

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

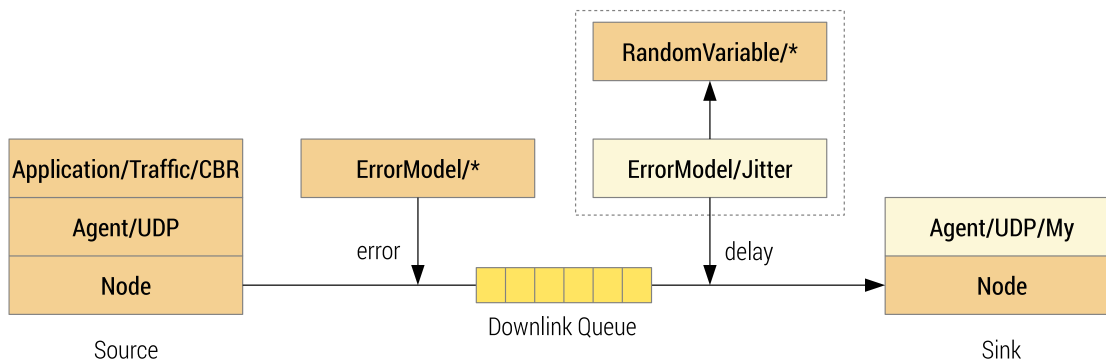


Figure 6.4: Simulation scenario for inverse modeling of jitter.

The last step before running the analysis is setting the objective. For this case study, we decided to use the average and standard deviation of the generated jitter. Informally, we require that the generated jitter remains within one standard deviation (JSD) from the average measured jitter (J), for at least 40 seconds (T). To reduce the influence of the random seed, 5 RTP transmissions are launched in parallel on different links and the objective is checked against the average jitter. This objective can be written as an LTL formula, using a similar approach to the objective used in Section 5.4.4 for validating an E-model implementation. More formally, we define this objective as:

$$\begin{aligned} ltlAccept: &\quad \Diamond(insideStdev \wedge (insideStdev\ U\ timeUp)) \\ def: &\quad timeUp\ :=\ time - startOf >= T \\ def: &\quad insideStdev\ :=\ inSd(avgJitter, J, JSD) \\ def: &\quad inSd(m,r,sd)\ :=\ (m \geq (r - sd/2)) \wedge (m \leq (r + sd/2)) \end{aligned} \tag{6.1}$$

The constants J and JSD, which represent the average and standard deviation of the jitter, are collected from the real field tests, and are defined separately for each field test scenario.

In order to perform the analysis efficiently, we apply a counter projection (see Section 4.2.1) to the ns-2 trace. First, we select the *time* and *avgJitter* variables to be present in the projected trace. The artificial variable *startOf*, which is computed every time a state is sent, is also projected. We also fold the projection with a time limit (see Section 4.2.3), to reduce the number of elements present in the projected trace. The rationale for doing this folded projection is the same as in the E-model validation case study of Section 5.4.

Finally, we perform the analysis with these scenarios, parameters and objective. The returned parameter configurations yield probabilistic distributions that match the real

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

Distribution	P.D.F.	Scenario	Parameters
Uniform	$\frac{1}{b-a}$ for $x \in [a,b]$	UMTS	a=0.001, b=0.029
		HSDPA	a=0.007, b=0.038
Exponential	$\lambda e^{-\lambda x}$	UMTS	λ=0.012626
		HSDPA	λ=0.013636
Normal	$\frac{1}{\sqrt{2\pi\sigma^2}}\, e^{-\frac{(x-\mu)^2}{2\sigma^2}}$	UMTS	μ=0.015, σ=0.01
		HSDPA	μ=0.0375, σ=0.01
Pareto	$\frac{\alpha x_{\mathrm{m}}^{\alpha}}{x^{\alpha+1}}$ for $x > x_m$	UMTS	x_{m}=0.001125, α=1.6
		HSDPA	x_{m}=0.0179, α=1.776

Table 6.1: Probability distributions for modeling jitter.

jitter, as imposed by the objective. Table 6.1 shows the parameters obtained for some combinations of scenarios and probability distributions. It is worth noting that there was no single "best" configuration on most scenarios, but rather a set of configurations that met the objective.

Figure 6.5 shows the four jitter distributions from Table 6.1 plotted against the jitter measured from one of the field tests. From the results it can be deduced that the characteristics of real jitter are best represented using a Pareto distribution.

6.4 Related work

There are different approaches in the literature for determining accurate jitter models. An interesting estimation of delay jitter based on a discrete-time queuing system is provided in [121]. The authors assume that jitter is due to multiplexing and buffering of voice and video packets at IP level with bursty background traffic. These conditions limit the applicability of the results obtained. For instance, new considerations should be introduced for cellular environments.

Like in our approach, the authors of [97] use field test traces, but as the input to a modified protocol stack to reproduce the loss and delay behavior in a test environment. In contrast, we use the traces to reproduce the characteristics observed in a realistic manner, not simply as a simple replay. Furthermore, they assume a network model based on the queue theory to parameterize the behavior observed. In our work we apply a higher level of abstraction because the model network is not established: instead, it is also abstracted by the mathematical expression we use to model the KPI.

The most remarkable contribution of our methodology, in comparison with previous work, is the abstraction capacity of the models obtained, which extends the applicability

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

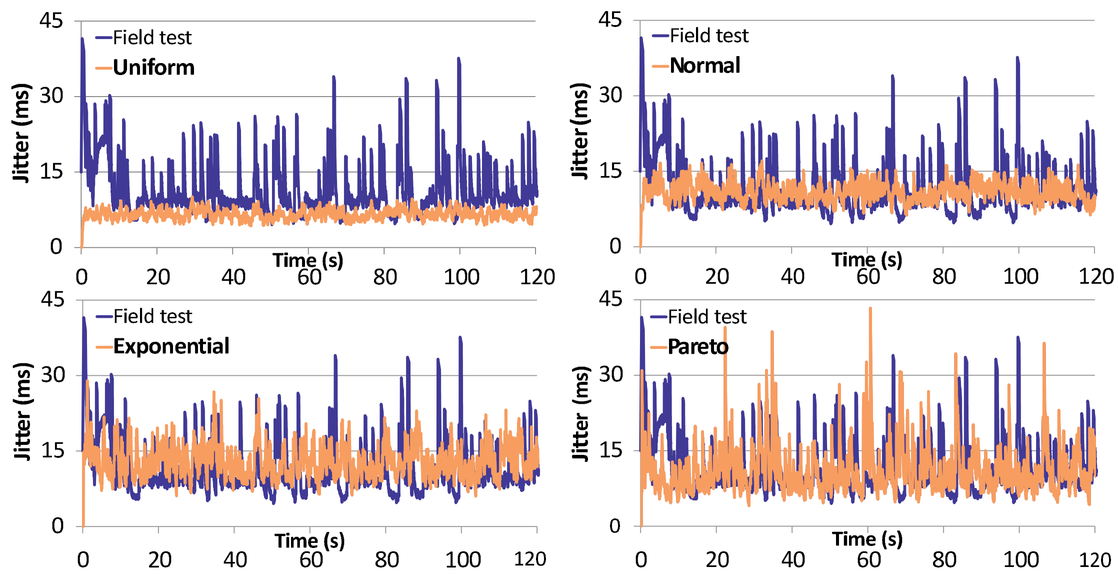


Figure 6.5: Comparison of different jitter models with a real jitter trace.

of the methodology to different scenarios without the need to introduce significant modifications. Also, the flexibility given by our communication protocol allows applying this methodology to other network simulators for which the protocol is imlpemented.

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

Chapter 7

Analysis of Java execution traces

Java is one of the most extended programming languages as of today. This language is used across a wide range of applications, from server programming to mobile applications for the Android operating system.

There are many developer productivity tools that have been developed for Java. These tools range from profilers to debuggers, and are geared towards improving the correctness and efficiency of Java programs. In Chapter 2 we have discussed some of these tools, in particular unit testing tools such as JUnit and TestNG that enable automated testing of programs according to a set of requirements, and model checkers such as JPF that can explore programs exhaustively. In this chapter we discuss how we can apply OptySim for testing Java programs, taking advantage of unique features such as checking LTL formulas on possibly infinite traces. We called this integration TJT.

The Java Virtual Machine (JVM) is the piece of software responsible for executing Java programs. Among its features, it provides a debugging interface which can be used to monitor and control the program being executed. We take advantage of this interface to extract the execution traces on-the-fly without modifying the original Java program.

7.1 The Java programming language

In order to analyze Java programs, the first step is defining what constitutes an execution trace and how to extract it. Java is a general-purpose, concurrent, class-based, object-oriented language [70]. Java programs are normally compiled into bytecode (an intermediate language), which is then interpreted by a Java virtual machine (JVM) in order to be executed. Java programs follow the imperative paradigm, where instructions are executed in sequence, each possibly changing the program state.

A Java program starts with a single execution thread, but it is possible to spawn more threads that will execute part of the program in parallel. Threads can be supported by one or more hardware processors, with or without time slicing. Each thread has its own

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

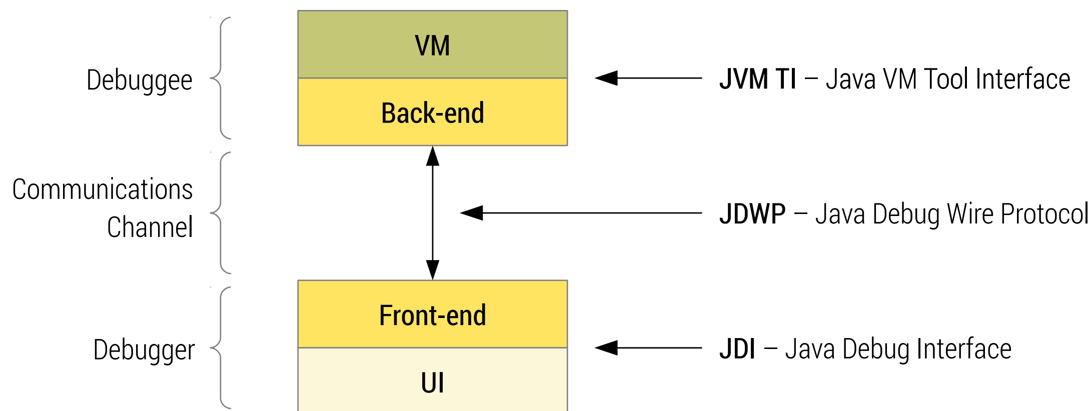


Figure 7.1: Overview of the Java Platform Debugger Architecture (JPDA)

call stack and program counter, but memory is shared among the threads contained in the same process.

In our framework, we consider that an execution trace is a sequence of states. In the case of the execution of a Java program, this sequence will be produced by the execution of the individual instructions. This definition is straightforward in the case of single-threaded programs, or multi-threaded programs running on a single processing unit. In the case of multi-threaded programs running on multiple processing units, we consider that an observer can only see the effect of one instruction at a time, even though the instructions may affect each other in incorrectly synchronized programs. The execution trace observed in this way should conform to the memory model as specified in the Java Language Specification, Section 17.4 [70].

7.1.1 Java Platform Debugger Architecture

Java defines a series of interfaces for debugging purposes, which together form the Java Platform Debugger Architecture (JPDA) [20]. Each interface is located at a different layer: JVM TI defines the debugging services provided by a JVM, JDWP defines the communication protocol between the debugger and debuggee processes, and JDI is a high level Java API for developing remote debugger applications. Figure 7.1 shows an overview of these components.

The Java VM Tool Interface (JVM TI) [4] is the lowest level debugging interface offered in JPDA. It is intended to be used by tools that need access to the VM state, such as debuggers and profilers. A client of JVM TI, called an agent, can be notified of events of interest in the JVM, and also control and query the application through the interface. Agents can be written in any native language that supports C language calling conventions. A JVM implementation is not required to provide JVM TI. However, higher level debugging interfaces may still be used by tools even if JVM TI is not available.

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

The next interface is the Java Debug Wire Protocol (JDWP) [19], which defines the communication protocol between the debuggee process and the debugger. This interface defines the messages that can be transferred between both parts. The transport mechanism for this protocol is not defined as part of the protocol. The reference implementation provides two transport implementations: TCP socket and shared memory. The back-end for the protocol can be implemented as a JVM TI agent, as in the reference implementation. JDWP provides roughly the same information as JVM TI, but with additional bandwidth-related functionality.

Lastly, the Java Debugger Interface (JDI) [18] is the highest-layer of JPDA. JDI is a high-level Java API for developing debugging tools that access the state of another JVM, usually remotely. This is the recommended interface for developing such tools, thanks to an API that is easier to use than connecting through JDWP or JVM TI directly.

JDI offers an event-driven interface, where the user can subscribe to receive relevant information. Supported events include: method entry and exit, modification of object fields, exceptions, monitor handling, thread lifetime, and arbitrary breakpoints. Using JDI, a tool can access the call stack and local variables of each thread, and query the objects allocated in the heap.

7.2 Extraction of Java execution traces

We took advantage of the features offered by the JDI API to develop a front-end for analyzing Java programs, which we called TJT [22][23]. In addition, we developed a graphical user interface as a plug-in for the popular Eclipse IDE [15]. The integration between our analyzer and Java was developed in parallel with our other applications. This led to similar, albeit ultimately incompatible, protocol implementation and features. Figure 7.2 shows an overview of this integration, split into the Eclipse plug-in and the analysis. This follows the general framework outlined in Chapter 3, with a few particularities.

7.2.1 Eclipse plug-in

The Eclipse plug-in is the main interface of the tool, and it assists the user in both writing test specification files and in reviewing their results. The plug-in provides a form-based editor for creating XML test specification files. Figure 7.3 shows an screenshot of the plug-in, with the test specification file editor. These files include the basic information described in Section 3.1.1, plus additional details such as breakpoints. This file is used to generate the code for a custom analyzer from a Promela template.

The test specification file editor also helps the user select the content of the projected states, by choosing from a list with all the variables present in the program to be analyzed. The user can also set breakpoints at specific code locations, and use them in LTL formulas.

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

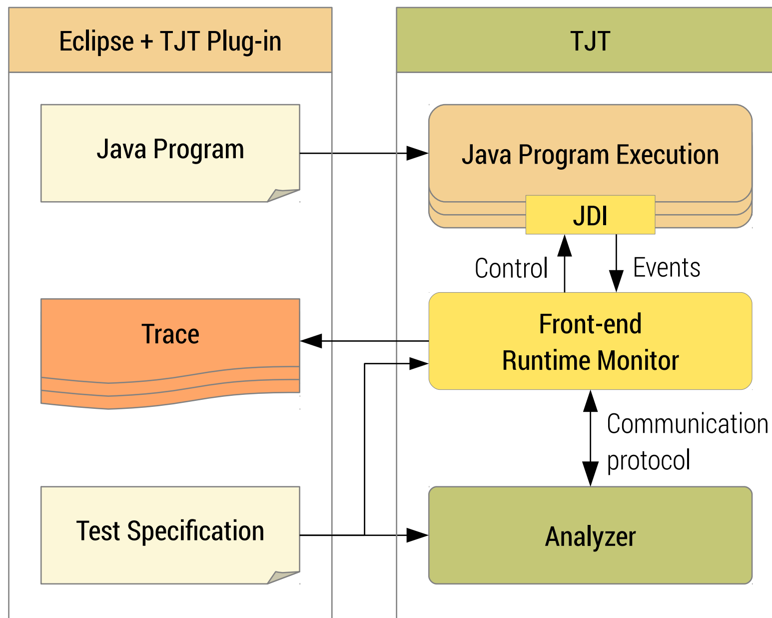


Figure 7.2: Overview of the analysis framework

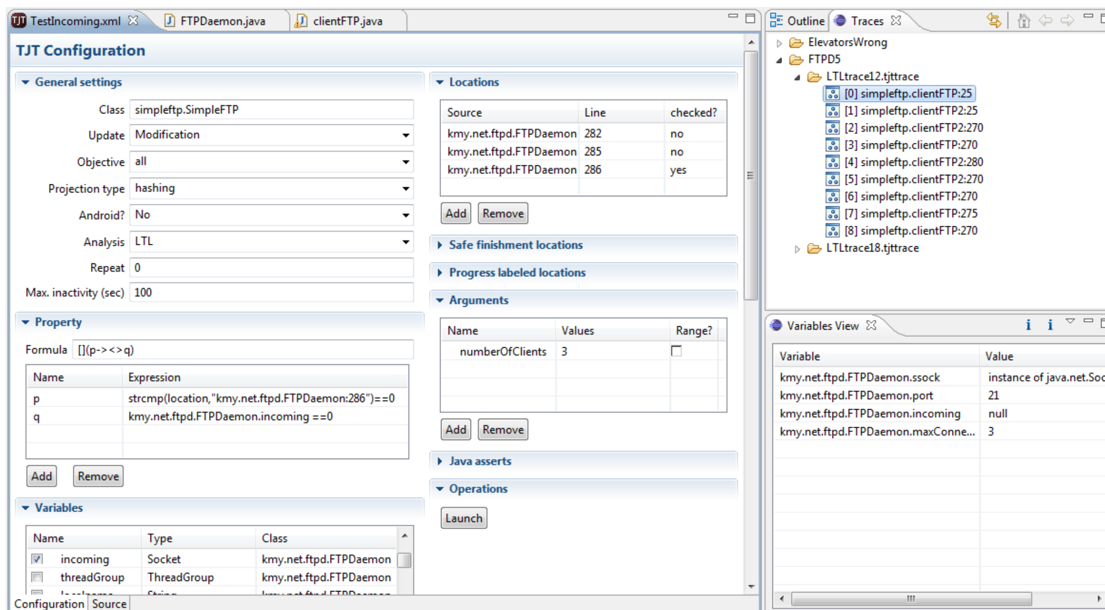


Figure 7.3: Screenshot of the TJT Eclipse plug-in

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

Since TJT is oriented towards testing and offering a friendly user interface, we decided to hide the classification of objectives from Section 3.3, and instead show the user a classification closer to that of traditional testing and model checking tools. In these cases, the user is typically only interested in traces that violate an objective, in order to locate bugs in the source code. The available tags are:

- ***all*.** The formula must be checked on every trace for the test to pass. This represents a desired behavior.
- ***none*.** The formula must not be checked in any of the traces for the test to pass. This represents an undesired behavior.
- ***any*.** It is enough for one trace to check the formula for the test to pass. Whether this represents a desired or undesired behavior is left to the user.

The fundamental difference with the regular OptySim tags is that these are assigned to the whole analysis after it has been completed, e.g. if at least one execution trace is found that does not met an *all* test, then the plug-in notifies the user that the test was not passed. Nevertheless, all execution traces are analyzed as usual.

The formula parser supports a few helper functions that can be used to refer to breakpoints and exceptions, among others:

- **loc(location).** Returns true if the program counter of the current state is located in the given source code location. Locations are given as strings with the format `"<file>:<line>"`, where the first part is the file and the second a line within that file.
- **exc(exception).** Returns true if the current state was produced by the exception whose class name is given as an argument.
- **streq(string, string).** Returns true if the given strings are equal.

Once the analysis finishes, the results are presented in a separate view in Eclipse. This view contains all the traces returned by the analyzer. For each trace, the projected states are shown, i.e. the states of interest that were sent to the analyzer. Clicking on any state opens the corresponding file and highlights the corresponding line of code in the editor view. In addition, the values of the variables in that state are shown in a separate view, helping the user locate the source of any objective violation.

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

7.2.2 Runtime monitor

The runtime monitor, which also acts as the front-end between the analyzer and the Java programs being analyzed, has been implemented in Java using JDI. The communication between the analyzer and the runtime monitor follows the general principles of the communication protocol described in Chapter 3, although the implementation is different and incompatible. The analyzer also includes additional state variables to hold information which is specific to this integration, such as the current location of the program counter, or the type of the last exception.

The runtime monitor is responsible for launching the Java program with the parameters sent from the analyzer, monitor the execution of the program, and send the relevant state updates back to the analyzer. The communication between the monitor and the analyzer happens through a TCP socket. The messages themselves are encoded using XDR [7], to ensure that the values sent over the socket are correctly represented on both sides.

JDI offers an event-based API, where the debugger can subscribe to some events of interest in the debuggee process. In this case, the runtime monitor subscribes to the following events:

- **Method entry and exit.** Each time the program enters or exits a method, an event is fired and captured by the runtime monitor. Classes from standard packages, e.g. `java.*`, are excluded by default.
- **Thread start and death.** The runtime monitor is notified whenever a thread is created and started, or a thread dies.
- **Synchronization.** Monitors are the most fundamental synchronization object in Java. The runtime monitor is subscribed to four different events related to monitors: a thread starts or finishes waiting on a monitor, a thread attempts to enter a monitor that is already acquired by another thread, and a thread enters a monitor after waiting for it to be released from another thread.
- **Field modifications.** The user can select certain instance fields that should be watched. When one of these fields is modified, an event is triggered and the monitor sends a new state to the analyzer.
- **Breakpoints.** When a thread reaches a specific location in the code, as set by the user, the runtime monitor is modified.
- **Exceptions.** The runtime monitor is notified when an exception happens

After any of these events, all the threads in the target JVM are suspended. This enables the runtime monitor to further inspect the program without any interference, if

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

required. For instance, in order to provide a hash of the current state of the JVM, as required for the hash projection (see Section 4.2.2), the objects in the heap and the thread stacks must be walked. This could not be done reliably while the program is still running.

This front-end for Java programs supports the two projections described in Chapter 4. The user can use the counter projection or the hash projection, if cycle detection is important for the test objectives.

7.3 Case studies

In this section we apply our tool to the testing of Java programs using requirements expressed as LTL formulas. As case studies we have used both publicly available open source programs and small custom examples. The former include three server applications for the HTTP, FTP and NFS protocols. The latter two were also evaluated in [60]. In addition, we developed two small examples: an implementation of a classical concurrency problem involving an elevator, and a infinite program that handles lists.

As is custom when testing server applications, we have developed mock clients to perform the tests in a controlled environment. These clients were implemented using only the essential behavior required for each test. In these cases, the main program that is executed is responsible for launching both the mock clients and the server.

We prepared a few tests for each case study and performed them using the hash projection, and also the counter projection where possible. Finally, we included a small comparison with JPF [74], a model checker for Java which was discussed in Chapter 2. The remainder of this section is dedicated to describing the programs that where tested, the tests performed on them, and the results of these analyses.

7.3.1 Applications and requirements

In this section we describe the Java applications that we have selected as case studies and the testing requirements we used as tests. These requirements have been expressed as LTL formulas, with the appropriate *all*, *none* or *any* tag. Table 7.1 contains these formulas, grouped by application. Note that all formulas, with the exception of `F4` and `F6`, represent liveness properties, and, in the case of programs with infinite executions, they can only be analyzed with runtime checkers that implement mechanisms like the cycle detection that enables the hash projection (see [115]).

FTP server

The first application is an FTP server [113]. This server accepts the usual FTP commands, such as `CWD` for changing the remote directory, or `CDUP` for navigating to the parent

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

Application	Formula		
FTP server	F1	all:	$\Box(\text{loc}(\text{"FTPDaemon:285"}) \rightarrow \Diamond(\text{incoming} = \mathit{null}))$
	F2	all:	$\Box\Diamond(\text{streq}(\text{FTPDConnection.status}, \text{"CDUP"}) \rightarrow \Diamond(\text{loc}(\text{"FTPDConnection:471"}))$
	F3	all:	$\Box((\text{req1} \rightarrow \Diamond(\text{resp1})) \wedge (\text{req2} \rightarrow \Diamond(\text{resp2})))$
	F4	none:	$\Diamond(\text{streq}(\text{FTPDConnection.status}, \text{"STOR"}) \wedge \neg\text{FTPDConnection.authentified})$
	F5	none:	$\Diamond((\text{incoming} = 0)\ U\ (\text{loc}(\text{"FTPDaemon:285"})))$
Elevator	F6	none:	$\Diamond(\text{loc}(\text{"Elevator:86"}) \wedge \neg\text{Elevator.isFree})$
	F7	all:	$\Diamond(\text{loc}(\text{"Elevator:86"}) \wedge \text{Elevator.isFree})$
NFS server	F8	any:	$\Diamond\Box(\text{MountdHandler.err} > 0)$
	F9	all:	$\Diamond(\text{MountdHandler.err} > 0 \wedge \Diamond\text{loc}(\text{"MountdHandler:95"}))$
	F10	none:	$\text{MountdHandler.err} = 0\ U\ \text{loc}(\text{"MountdHandler:95"})$
Web server	F11	any:	$\Box(\Diamond(\text{loc}(\text{"WebServerMain:63"}) \wedge \Diamond\text{exc}(\text{"WebServerException"})))$
	F12	all:	$\Box\neg\text{loc}(\text{"Client:41"})$
Lists	F13	all:	$\Box\Diamond(\text{numElements} = 1)$

Table 7.1: LTL formulas used as testing objectives

directory. The server settings are loaded from a file, which includes a list of users and their respective home directories.

The core of the server is the loop shown in Listing 7.1. The server accepts client connections iteratively and spawns a new thread, an instance of `FTPConnection`, to handle each client. This class contains methods that perform the required actions for each FTP command. For instance, part of the actions related to the `CWD` command are shown in Listing 7.2, in the `FTPConnection.CWD()` method.

We used formulas F1, F2, F3, F4 and F5 from Table 7.1 as testing objectives. Formula F2 was prepared to uncover an error that was purposefully introduced in the server code, while the rest of the formulas were tested on the original code. We now describe briefly the purpose of each of these tests:

- **F1.** After reaching line 285 of the `FTPDaemon.java` file (see Listing 7.1), the `incoming` variable should be `null` at some point in the future. This test is to ensure that `incoming` is properly cleared after each iteration of the server.

- **F2.** If a client tries to escape its home directory by performing repeated `CDUP` commands, the server should end up throwing, but capturing, an exception, and letting the client know that an error happened. The `CDUP` command is implemented as `CWD` operation, shown in Listing 7.2. The `FTPException` provoked by an erroneous `CDUP` command is captured and handled in line 471 of said listing.

- **F3.** A client may have two of its commands consecutively processed by the server,

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

```
273 // FTPDaemon.java
274 public void run() {
275     while(true)
276         try {
277             Socket incoming;
278             ServerSocket ssock1;
279             synchronized (this) {
280                 if (ssock == null)
281                     init();
282                 ssock1 = ssock;
283             }
284             incoming = ssock1.accept();
285             FTPDConnection conn = createConnection(incoming);
286             notifyListeners(conn, "");
287             conn.start();
288         }
289         catch(Exception e) {
290             e.printStackTrace();
291         }
292 }
```

Listing 7.1: FTP server: main loop

while another client with a pending request is starved. This test deals with the fairness in scheduling tasks. Multithreaded programs are prone to fairness issues: some threads may take all available CPU time, leaving other starving. This test checks whether this is a possible outcome under the default scheduling employed by the JVM or under other schedulings that may be enforced in the execution. The boolean propositions in the formula, such as *req1*, are defined in terms of auxiliary boolean variables in the mock FTP clients. This is also in contrast with other tests for this application, which deal with variables or locations in the FTP server itself.

- **F4.** No user should be able to perform a `STOR` command before being authenticated first.

- **F5.** At some point, the variable `incoming`, which holds the socket just opened for attending a new client, should be non-null until a new thread is spawned for attending that client. This formula is similar to F1, but using the "until" temporal operator (U).

Concurrency problem: elevator and clients

This application is a typical example in concurrency: a shared resource and several clients which try to access it at the same time. In this case, the shared resource is an

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

```
461 // FTPDConnection.java
462 public String CWD(String dir) throws IOException {
463     try {
464         String newDir = makePath(dir);
465         VirtualObject obj = server.produceObject(newDir, user);
466         Reader annot = obj.getAnnotationReader();
467         path = newDir;
468         return "250 CWD command successful.";
469     }
470     catch (FTPDException e) {
471         return "550: " + e.getMessage();
472     }
473 }
```

Listing 7.2: FTP server: CWD command

elevator, and the only operation allowed is riding it from a floor to another one. While clients can send requests at any time, only one client is allowed to ride the elevator at a time. Listing 7.3 shows part of the code of the `Elevator` class, in which the elevator first waits for a client request, and then moves to the floor where the client is located.

We tested this applications using the objectives set by formulas F6 and F7:

- **F6.** The elevator does not wait for clients when it is not free. That is, in no execution (*none*) the elevator should not wait on the condition that is shared with the clients, if there is one client that has requested to use the elevator. For the sake of illustration, this test was performed on the original code, as well as on a version with a bug purposefully introduced in the condition on line 84.
- **F7.** This formula is the opposite of formula F6, in the sense that it requires that *all* executions conform to the correct condition.

NFS server

We also performed some tests on a NFS server implemented with Java [103]. In particular, we tested some error conditions when mounting a remote file system, which is the handled by class `MountdHandler`. Formulas F8, F9 and F10 test several aspects of this functionality:

- **F8.** Clients cannot access incorrect or unauthorized directories. We use mock clients that perform erroneous requests, and check the value of an error field of the server.
- **F9.** This formula describes a testing requirement similar to F8, but using other temporal operators and checking the program location where the error should be detected.

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

```
82 protected void waitForClient() {
83     l.lock();
84     while (!isInterrupted() && isFree) {
85         try {
86             waitingForClient.await();
87         }
88         catch (InterruptedException e) {
89         }
90     }
91     int distance = Math.abs(currentFloor - requestedFloor);
92     while (distance != 0) {
93         Thread.sleep(150);
94         distance--;
95     }
96     waitingForArrival.signal();
97     l.unlock();
98 }
```

Listing 7.3: Elevator: elevator waiting for clients

- **F10.** In line with previous formulas, no other errors should happen before this one.

This server features an infinite loop where client requests are attended. In order compare the use of our two execution trace projections, we prepared a modified version of the server without this infinite behavior so it could be checked with the counter projection.

Web server

Finally, we also studied the Jibble Web Server [96]. As usual, this server contains a main loop which accepts client connections, spawning a new thread for each connection. In order to compare the performance of both of our trace projections, we also prepared a version with accepts a limited number of connections in order to provide a finite trace.

We used formulas F11 and F12 as testing objectives:

- **F11.** Launching the web server with an invalid root directory should not be allowed. We check that the appropriate exception is thrown in this case.

- **F12.** The server should be able to start properly on any of the ports of a given range. This range is specified in the test specification file. The formula checks that mock client are able to connect successfully to the server on any of these ports or, more precisely, that in all execution traces, clients are not unsuccessful when connecting to the server.

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

7.3.2 Counter projection

In this section we describe the tests carried out using the folded counter projection (see Section 4.2.1). It is worth noting that not all of the proposed tests can be carried out using this projection. Formulas that would require cycle detection cannot be checked, as per Proposition 4.2. However, some formulas can be checked on infinite traces generated from finite ones using Spin's stuttering mechanism, if the cycle created by repeating the last state infinitely is the only one required by the corresponding never claim automata. Most of the programs we are debugging are infinite, i.e. they are servers with an infinite reactive loop, and thus cannot be checked with finite resources and this projection. In some cases, as noted in the previous section, we modified these programs to produce finite versions that could be checked with both projections, for comparison. The results where the servers were modified in this way are marked.

The results are summarized on the left hand side of Table 7.2, averaged over a series of test executions. The third column shows the number of projected Java states, while the fourth column indicates the number of state transitions in Spin. The next two columns show the size of a Spin state and the total time of analysis. The last column of the table shows an approximation of the size of the Java states, before any projection. This number only takes into account the size of objects allocated in the heap. Since the size of the heap changes dynamically, we report the maximum value the we observed during the execution of each program.

The size of states in Spin after the projection is influenced by several factors. First, Spin has an overhead of 16 B for a Promela specification with a single process, and a Büchi automata adds another 8 B. Then, a `step` integer variable is added to track the Java state that is retrieved in each state (see Section 3.4.2). The variables used for generating test inputs, if any, are added to the global state as well. Furthermore, the counter projection requires an additional variable as described in Section 4.2.1. It is worth noting that the variables being monitored are not part of the global state, but are kept in a separate data structure, in order to support backtracking with minimal impact in the size of Spin states.

TJT also has a deadlock detection algorithm. Although the purpose of formula `F7` was to detect the incorrect wait condition, it uncovered a deadlock in each execution of the program detected by the monitoring module. Although deadlocks may be detected while using any property, omitting the temporal formula is recommended when specifically searching for them in order to prevent the trace being terminated early due to a specified property.

7.3.3 Hash projection

We also evaluated the hash projection, using all the properties described in Section 7.3.1. Thanks to its cycle detection capabilities, we could use it for more tests than the counter

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

		Counter projection				Hash projection				
Application	Formula	States	Trans.	S. size	Time	States	Trans.	S. size	Time	J. size
FTP server	F1	-	-	-	-	47	83	76 B	4.2 s	4.27 MB
	F2[3]	-	-	-	-	150	187	76 B	14.9 s	
	F3	-	-	-	-	2419	6211	76 B	34.7 s	
	F4	-	-	-	-	59	265	76 B	17.6 s	
	F4[4]	130	131	48 B	0.8 s	171	172	76 B	0.9 s	
	F5	-	-	-	-	33	34	84 B	3.9 s	
	F5[4]	19	20	48 B	0.8 s	23	24	76 B	5.5 s	
Elevator	F6	31	219	40 B	28.7 s	31	204	68 B	34.7 s	0.74 MB
	F6[3]	3	54	40 B	23.0 s	3	54	68 B	24.1 s	
	F7[3]	3	51	40 B	29.0 s	3	51	68 B	33.7 s	
NFS server	F8[4]	15.3	94.3	40 B	5.1 s	17.6	99	68 B	10.8 s	4.08 MB
	F9[4]	5	140	40 B	22.9 s	5	140	68 B 1	27.1 s	
	F10	4	14	40 B	4.6 s	4	14	68 B	5.2 s	
Web server	F11	-	-	-	-	211	231	92 B	202 s	6.71 MB
	F11[4]	165	175	64 B	8.6 s	258	270	92 B	13 s	
	F12	-	-	-	-	135	146	96 B	214 s	
	F12[4]	168	179	68 B	9.1 s	273	284	96 B	13.5 s	
Lists	F13	-	-	-	-	6	33	68 B	0.7 s	0.14 MB

Table 7.2: Test results using folded counter and hash projections

projection. The right half of Table 7.2 shows these results. The table shows that, compared with the counter projection, the hash projection is generally slower. This can be attributed to the computation penalty associated with visiting the whole Java program state and computing its hash. As these results suggest, the tests with more Java states are the ones where the test time increases the most. Also, the difference between the size of Spin states between the counter and the hash projections is constant: the hash projection adds a variable to store the hash of each state, but removes the counter variable. Although the size of the hash proper is 16 B, in our implementation it is stored as a 32 B character array, which explains the total difference of 28 B.

In addition, we also tested a simple Java program that deals with lists in an infinite loop, adding elements to the list first and then removing them. We checked that the list ended up with exactly one element an infinite number of times, which is formalized in formula F13.

[3]Formula tested over a program where errors were manually introduced.
[4]Formula tested over a finite version of the original infinite program.

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

		TJT (Hash projection)		JPF-LTL	
Example	Formula	Transitions	Time	Transitions	Time
Eventually2	$\Diamond done()$	8	0.6 s	10	0.6 s
AlwaysEventually1	$\Box\Diamond done()$	68	2.2 s	8	0.5 s
DoubleEventuallySwitch1	$\Diamond done() \vee \Diamond foo()$	8	0.6 s	37	0.6 s

Table 7.3: Results of tests using TJT and JPF-LTL

7.4 Related work

For the integration of our OptySim analysis framework and Java, we have focused on reliability analysis in general, and testing in particular. Section 2.2.2 described some tools for Java which have a similar target, focusing on model checking and testing tools.

7.4.1 Comparison with JPF

As we explained in Section 2.2.2, JPF [74][117] is a model checker for Java programs. It performs an exhaustive exploration of a Java program using a modified JVM. Although JPF does not support LTL properties like OptySim, there is an ongoing effort to add support for them via JPF extension mechanisms [90]. However, it does not currently support as many propositions as OptySim. On the other hand, OptySim does not perform an exhaustive analysis of the whole state space of a program, but instead studies a significant set of execution traces.

Thus, even though both tools are based on model checking, they have different scopes. However, in this section we provide a small comparison between the two tools, based on examples provided with the JPF-LTL extension. In order to perform this comparison, we used only the hash projection, as it supports the cycle detection required in all the examples. The results are summarized in Table 7.3. In this table, the first two columns show the name of the example and the LTL formula being used, "Transitions" shows the number of state transitions traversed, and "Time" the total time required to check the formula against the program. As noted in Section 2.2.2, the atomic propositions in these formulas reference method entry, which both JPF-LTL and TJT support.

It is worth noting the disparity in time and space required for the analysis of the second formula with TJT, compared to the other two. This program deals with random number generators, and the property requires cycle detection. Although checking whether or not a single trace violates the property is relatively quick, the first few traces generated and analyzed by TJT did not violate the property. Thus, when a violating trace was generated, the cost of the analysis had accumulated the analysis of the previous traces.

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

Part IV

Final remarks

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

Chapter 8

Conclusions and Future Work

In previous chapters we have outlined the problem we aimed to solve, namely the definition of a single framework for analyzing execution traces using model checking. We tried to show its usefulness through its application to different system and analysis types. In this chapter we summarize our work and reflect on our contributions. Finally, we show how this thesis could be extended in the future.

8.1 Conclusions

Many techniques and tools have been proposed for the analysis of hardware and software systems. However, most of them are geared towards a specific type of system and/or analysis, forcing users to learn different tools and formalisms. In this thesis we have presented an approach that uses model checking to analyze external systems whose behavior can be observed as execution traces. This single framework, called OptySim, allows the user to apply the same kind of objective-driven analysis to different kinds of systems.

Although it is based on model checking, our framework does not perform full state space exploration of the external systems, but rather of a set of execution traces from the system. However, we believe that this approach is useful for many situations, as our case studies have shown. For instance, when used for testing OptySim cannot guarantee the absence of errors in a system as a model checker could. On the other hand, it may be able to analyze a significant portion of systems whose size make them too big to be analyzed exhaustively.

OptySim separates the analyzer, implemented using the Spin model checker, from the system being analyzed. Both components are connected using a communication protocol which includes primitives to set up executions and receive the required execution traces. This separation requires a front-end to be developed for each type of system to be supported. We have implemented such front ends for Java programs and the ns-2 network

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

simulator. The latter has been implemented in C++, in a way that can be extended for other C++-based systems.

This separation enables the evolution of both components independently. The analyzer could be connected with more types of systems by developing an appropriate front-end that is able to set-up executions and extract execution traces. Our communication protocol could also be reused by other tools that perform other kinds of analyses over execution traces.

The analyzer has been implemented using a Promela template that is instantiated for each particular analysis. For instance, the variables selected by the user as part of the trace have to be included in the global state. This template is supported by additional C functions and libraries that handle communication, state meta-data, and trace storage and retrieval, among others. The latter is instrumental in supporting proper exploration of the state space within a trace. Although traces are fixed sequences of states, a property encoded as a never claim automata may require backtracking. Trace storage support ensures that trace history is backtracked properly, while allowing the reception of new states.

User-defined objectives serve as a guide for the analysis, while providing meaningful results for the user. On the one hand, the results indicate which objective was met for each trace, using tags. On the other hand, the analysis of a single trace can be terminated early if the user chooses it, saving resources when a single answer per trace is enough. This is thanks to our execution trace-based exploration semantics and the on-the-fly reconstruction of execution traces.

Since our framework is based on Spin, it would be possible to add support for more formalisms to describe objectives easily. For instance, Spin supports correctness requirements expresses as timelines of events, that are then translated into never claim automata.

We have applied OptySim to different types of analysis, in a series of case studies described in Part III. First, we studied the performance of a video over TCP application in ns-2, and used the results to build an optimized version of the application. The definition of the parameter space and the objectives that guide the analysis allows us to explore a great number of scenarios from which to extract performance results. We also tested the correctness of the models used in this scenario.

Unfortunately, the integration with ns-2 is not completely transparent from the point of view of the designer of the network scenarios. The extraction of the execution traces may require some support from the user, e.g. additional internal probes may be needed in order to extract relevant variables. This is in part due to the philosophy of ns-2, where most analysis are performed over packet traces, and not all network elements have made their internal state traceable.

We also prepared a validation scheme for E-model implementations. This scheme was based on comparing several metrics computed from both the implementation being

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

validated and a reference implementation. A similar approach was used to tune KPI models using data collected from field tests. We applied this approach to obtain jitter models that replicated the characteristics observed in real conditions.

Our case studies analyzing Java programs where centered around testing. We used several existing Java server applications, plus two synthetic examples, as case studies. The Java integration supports both the counter and hash projections, and we compared them in some of the tests. While the counter projection was clearly more efficient, the hash projection enabled us to write tests which could not be expressed otherwise. The integration with Java programs, in contrast with ns-2, is completely transparent thanks to the use of the JDI API.

To summarize, we believe our approach has proven to be a valid alternative to other analysis techniques. The ability to analyze external systems through their execution traces is a useful solution for many kinds of analysis, and for systems that could not be studied otherwise. Furthermore, the proposed communication protocol facilitates the integration with many types of systems under a unified framework.

8.2 Future work

The approach presented in this thesis is open in many aspects. We believe that the following lines of work are worth pursuing.

Integration with the ns-3 network simulator When this thesis was started, ns-2 was the *de facto* standard network simulators among most academic papers. ns-3 [77] had already begun development and had a promising architecture, but it was lacking the breadth of modules available for ns-2. Nowadays, however, ns-3 has catch up with ns-2, and newer technologies such as LTE are better supported in the former [101]. Since the core of ns-3 is still implemented using C++, the integration library described in Section 5.2 can be adapted for ns-3 by implementing a new `Engine` subclass with the appropriate hooks. For instance, ns-3 provides a complete attribute system that can be leveraged for both parameter configuration and trace extraction.

Integration with model-driven tools Model-driven Engineering is a popular choice for developing software systems. In the field of protocol stacks and network software, it has been supported by tools such as Telelogic SDL Suite [16] and Rational Rhapsody [17] (both now owned by IBM). A natural continuation for our work with ns-2 would be to integrate with these tools, offering analysis capabilities at the model level, e.g. setting up parameter configurations and objectives using the artifacts present in the models. In the case of Rational Rhapsody, the final C++ source code can be used as the integration point, by leveraging the C++ integration library once more. However, the tool should perform a

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

translation between model and source code artifacts, so that the user does not have to dive in the generated code to set up the analysis and review its results. This translation would be similar to the one performed by the animation capabilities provided by Rhapsody: the final source code is executed, but additional instrumentation is included to identify the model artifacts that are being used.

Wider hash projection support The hash projection described in Chapter 4 has only been implemented for the Java integration (see Chapter 7). The Java JDI API was be used to walk the complete program state, i.e. both stack and heap memory, in order to compile a hash for the projection. This is potentially harder to do in other non-managed languages, such as the combination of C++ and OTcl employed by ns-2. However, having the option of using this projection with more types of systems would expand the set of possible objectives, thanks to the support for cycle detection.

Support more formalisms for objectives In Section 3.3 we describe the types of objectives which OptySim supports, mainly state asserts, LTL formulas and never claim automata.Although LTL formulas and never claim automata allow the definition of complex objectives, they may not be easy to use and comprehend for users that are experts on their own domain, but not model checking. It would be possible to add support for more formalisms that are more amenable to users by providing a suitable translation to one of the supported ones. For instance, network protocol designer could write requirements as message sequence charts (MSCs) to be analyzed by OptySim.

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

Part V

Appendices

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

UNIVERSIDAD DE MÁLAGA
Publicaciones y Divulgación Científica

Appendix A: Configuration files

This chapter describes the files used to configure and perform the analysis of a system. These files include information such as the definition of the parameter space, the objectives and the variables that should be part of the projected trace.

The definition of these files has evolved with the thesis itself and its applications. Here we show the latest version of the configuration files, defined as an XML Schema, although legacy versions can still be found in some examples.

A.1 XML Schema

Listing A.1 contains the complete definition of the XML Schema for configuration files. The only element defined in the schema is the root element, `optysimconfig`, which is of type `configuration`, while the rest of the schema is defined using nested elements and types. An overview of the schema and the most important elements is included below.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <schema targetNamespace="http://www.lcc.uma.es/~salmeron/optysim"
       elementFormDefault="qualified"
       xmlns="http://www.w3.org/2001/XMLSchema"
       xmlns:tns="http://www.lcc.uma.es/~salmeron/optysim"
       xmlns:jaxb="http://java.sun.com/xml/ns/jaxb" jaxb:version="2.0">
3
4    <!-- Root element -->
5
6    <element name="optysimconfig" type="tns:configuration"></element>
7
8    <complexType name="configuration">
9      <sequence>
10       <element name="properties" type="tns:properties" maxOccurs="1"
            minOccurs="0" />
11       <element name="parameters" type="tns:parameters" maxOccurs="1"
            minOccurs="0" />
12       <element name="variables" type="tns:variables" maxOccurs="1"
            minOccurs="0" />
13       <element name="options" type="tns:options" maxOccurs="1"
            minOccurs="0" />
```


Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

```
14      </sequence>
15    </complexType>
16
17    <!-- Properties -->
18
19    <complexType name="properties">
20      <sequence>
21        <element name="ltl" type="tns:ltl" maxOccurs="unbounded"
              minOccurs="0" />
22        <element name="similar" type="tns:similar"
              maxOccurs="unbounded" minOccurs="0" />
23        <element name="never" type="tns:never" maxOccurs="unbounded"
              minOccurs="0" />
24        <element name="other" type="tns:otherProperty"
              maxOccurs="unbounded" minOccurs="0" />
25        <element name="define" type="string" maxOccurs="unbounded"
              minOccurs="0" />
26      </sequence>
27    </complexType>
28
29    <complexType name="property">
30      <simpleContent>
31        <extension base="string">
32          <attribute name="name" type="string" use="required" />
33          <attribute name="type" type="string" />
34        </extension>
35      </simpleContent>
36    </complexType>
37
38    <simpleType name="propertySubtype">
39      <restriction base="string">
40        <enumeration value="accept" />
41        <enumeration value="reject" />
42      </restriction>
43    </simpleType>
44
45    <complexType name="ltl">
46      <complexContent>
47        <extension base="tns:property">
48          <attribute name="formula" type="string" use="required" />
49          <attribute name="subtype" type="tns:propertySubtype"
                use="required" />
50        </extension>
51      </complexContent>
52    </complexType>
53
54    <complexType name="similar">
55      <complexContent>
56        <extension base="tns:property">
```


Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

```
57        <attribute name=”formula” type=”string” use=”required” />
58        <attribute name=”subtype” type=”tns:propertySubtype”
             use=”required” />
59      </extension>
60    </complexContent>
61  </complexType>
62
63  <complexType name=”never”>
64    <complexContent>
65      <extension base=”tns:property”>
66        <attribute name=”subtype” type=”tns:propertySubtype”
             use=”required” />
67      </extension>
68    </complexContent>
69  </complexType>
70
71  <complexType name=”otherProperty”>
72    <simpleContent>
73      <extension base=”tns:property”>
74        <attribute name=”formula” type=”string” />
75      </extension>
76    </simpleContent>
77  </complexType>
78
79  <!-- Parameters -->
80
81  <complexType name=”parameters”>
82    <sequence>
83      <element name=”parameter” type=”tns:parameter”
           maxOccurs=”unbounded” minOccurs=”1” />
84    </sequence>
85  </complexType>
86
87  <complexType name=”parameter”>
88    <complexContent>
89      <extension base=”tns:variable”>
90        <sequence>
91          <element name=”value” type=”string” maxOccurs=”unbounded”
               minOccurs=”0” />
92          <element name=”range” type=”tns:range”
               maxOccurs=”unbounded” minOccurs=”0” />
93        </sequence>
94      </extension>
95    </complexContent>
96  </complexType>
97
98  <complexType name=”range”>
99    <sequence>
100     <element name=”start” type=”string” maxOccurs=”1”
```


Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

```
          minOccurs="1" />
101       <element name="end" type="string" maxOccurs="1" minOccurs="1"
          />
102       <element name="stepSize" type="string" maxOccurs="1"
          minOccurs="0" />
103     </sequence>
104     <attribute name="numValues" type="int" />
105   </complexType>
106
107   <!-- Variables -->
108
109   <complexType name="variables">
110     <sequence>
111       <element name="variable" type="tns:variable"
          maxOccurs="unbounded" minOccurs="1" />
112     </sequence>
113   </complexType>
114
115   <complexType name="variable">
116     <attribute name="name" type="string" />
117     <attribute name="type" type="tns:varType" />
118     <attribute name="result" type="boolean" default="false" />
119   </complexType>
120
121   <simpleType name="varType">
122     <restriction base="string">
123       <enumeration value="char" />
124       <enumeration value="short" />
125       <enumeration value="int" />
126       <enumeration value="long" />
127       <enumeration value="float" />
128       <enumeration value="double" />
129       <enumeration value="time" />
130       <enumeration value="bw" />
131       <enumeration value="size" />
132     </restriction>
133   </simpleType>
134
135   <!-- Options -->
136
137   <complexType name="options">
138     <sequence>
139       <element name="option" type="tns:option" maxOccurs="unbounded"
          minOccurs="0" />
140     </sequence>
141   </complexType>
142
143   <complexType name="option">
144     <simpleContent>
```


Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

```
        <extension base="string">
          <attribute name="name" type="string" use="required" />
        </extension>
      </simpleContent>
    </complexType>

</schema>
```

Listing A.1: XML Schema definition of configuration files

The main types of the schema are:

configuration The type of the root element of the document. It contains at most one of the following nested elements: `properties`, `parameters`, `variables` and `options`.

properties A sequence of one or more `property` elements, each representing a property or objective. A `property` can be one of: `ltl`, `similar`, `never`, `otherProperty`, `define`.

ltl An LTL formula written as a string, using the notation accepted by Spin, and a `accept` or `reject` subtype.

similar A boolean formula using the current and previous values of the parameters, and a `accept` or `reject` subtype. While current parameters are prefixed with a dollar sign, previous values are not, e.g. `$param1 == $param1`. `$*` can be used as a shorthand to indicate that parameters which have not been mentioned in the formula should have the same value.

never The path to a file containing a never claim automata, using the notation accepted by Spin.

otherProperty Another kind of property, where the `type` attribute further identifies its kind. Current valid values of `type` are: `invalid_config` (a boolean condition to discard configurations before they are executed), `assert` (simple asserts checked after each step, for debugging purposes), `accept`, `reject` (like asserts, but produce the corresponding result).

define A macro definition, e.g. used in a property, to be placed in the generated Promela file.

parameters A sequence of one or more `parameter` elements, each representing a parameters of the system and its possible values. The values of each `parameter` are defined by the combination of one or more `values` (several discrete values) or `range` (a range with start, end, and either step size or number of steps).

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

`variables` A sequence of one or more `variable` elements, each representing a variable of interest in the system that should be present in the projection.

`options` A sequence of one or more `option` elements, which are key-value pairs containing miscelaneous options, some of which may be particular to some systems or template instantiations.

Supported options

The tools currently supports the following options:

`filename` The path to the main file of the system, used in its execution. This option may represent an executable filename (for UML/C++ and ns-3) or a scenario description filename (for ns-2).

`printVars` A boolean value. For ns-2 simulations, if true, print system variables after each step.

`tracePackets` A boolean value. For ns-2 simulations, if true, also generate packet trace files for each simulation.

A.2 Legacy configuration

Although the previous XML Schema is the new and preferred way of configuring analyses, many examples still use older methods which are still supported. This section describes two such methods briefly: configuration files for ns-2 scenarios and for TJT (i.e. Java).

A.2.1 ns-2 legacy configuration

Originally, configuration options for ns-2 scenarios were embedded as comments in the scenario files. All of the following configuration declarations are comments, thus prefixed with a hash symbol (#).

The configuration is split into several sections: *config*, *measures*, *properties*, *defines*, *options* and *model*. Section names must be enclosed in brackets, e.g. `[config]`. The purpose of each section as follows:

`config` System parameters for generating configurations. Each parameter must declare its type, and include a type a series of values or a range, but not both.

`measures` The variables of interest of the ns-2 simulation that must be present in the projection.

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

`properties` Objectives for the analysis. The accepted objectives are the usual, although there are additional legacy aliases, e.g. *optimum* for *accept*, and *bound* for *reject*.

`defines` Definition of macros and constants, to be used in objectives.

`options` Additional options given to the analyzer. The original implementation did not define any for ns-2, instead using command-line options given to the template instantiator.

`model` The ns-2 scenario itself. No special declarations are expected in this section.

A.2.2 TJT legacy configuration

The configuration files for the Java integration are also defined by a XML Schema. This schema is similar, although incompatible, with the one described in Section A.1. The main top-level elements of this schema are:

class The name of the class which contains the `main()` method which starts the program.

property An LTL formula to be used as objective.

literal A list of macro and constant definitions to be used in objectives.

repeat The number of times each parameter configuration shoul be executed and analyzed.

update With this option the runtime monitor can notify the analyzer on method entry and exit, as well as when a variable of interest is modified.

objective Whether the LTL formula should be interpreted as a *all*, *any* or *none* objective.

projection The trace projection to use: counter or hash.

variableList A list of instance fields which should be monitored and included in the projected execution trace.

locationList A list of breakpoints that should trigger a new state in the execution trace.

argumentList A list of program parameters for generating configurations. Each parameter includes a range of valid values.

android Indicates that the program to be analyzed is an Android app, which requires special handling.

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

Appendix B: Protocol definitions

This chapter contains a more complete definition of the communication protocol described in Section 3.2.

B.1 Promela model

The activity diagrams shown in Figures 3.3 and 3.4 of Section 3.2 contained an overview of the communication protocol. However, these diagrams excluded some details for illustration purposes, such as error handling. In this section we present a Promela model of the protocol.

Protocol messages have been simplified to include only a type and a parameter, defined using the `mtype` declaration. Only the key parameters have been defined. For instance, the type of a *Report* message is defined, but not the particular contents in the case of a *Report(Measure)*.

This Promela model defines four processes: analyzer (`analyzerProc`), system (`systemProc`), and two processes for simulating errors in protocol implementation (`upstreamChannel` and `downstreamChannel`). With the exception of the system process, all processes are started at initialization. `analyzerProc` contains a outer `do` loop, which models each time the analyzer executes the system using a new parameter combination. The system process is started by the analyzer process at the start of each iteration. This loop can terminate at any point, i.e. because the parameter space has been exhausted. `upstreamChannel` and `downstreamChannel` forward messages to the analyzer and system, respectively. In addition, they can change the type of the message and/or its parameter, to simulate bad analyzer or system implementations.

```
1 // Messages
2 mtype = { FeaturesDeclaration, FeaturesNegotiation,
3     VariablesDeclaration, VariableSelection, ParameterSetup,
4     Report, Command,
5     WrongMessage }
6
7 // Message parameters
8 mtype = { NoneAny,
9     OneTimeNegotiation, OneTimeSetup, SkipVariablesDeclaration,
```


Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

```
10      OTN_OTS, OTN_SVD, OTN_OTS_SVD, OTS_SVD,
11      Ok, Cancel,
12      Start, Stop, Reset, CommandString,
13      Ready, Started, Measure, Error, Stopped, Finished,
14      WrongParameter }
15
16  // Global channels: { Message, Message parameter }
17  chan toAnalyzer = [1] of { mtype, mtype }
18  chan toSystem = [1] of { mtype, mtype }
19  chan analyzer = [2] of { mtype, mtype }
20  chan system = [2] of { mtype, mtype }
21  chan analyzerCtrl = [1] of { mtype, mtype }
22  chan systemCtrl = [1] of { mtype, mtype }
23
24  // Negotiation options
25  bool oneTimeNegotiation = false;
26  bool oneTimeSetup = false;
27  bool skipVariablesDeclaration = false;
28  bool negotiationDone = false;
29  bool setupDone = false;
30
31  // Forwards channel "toSystem" to channel "system"
32  active proctype upstreamChannel() {
33    mtype m, p;
34
35  end:
36    do
37    :: atomic { toSystem?[m, p] && nfull(system) ->
38        toSystem?m, p;
39        if
40        :: system!m, p
41        :: system!m, WrongParameter
42        :: system!WrongMessage, WrongParameter
43        fi
44      }
45    od
46  }
47
48  // Forwards channel "toAnalyzer" to channel "analyzer"
49  active proctype downstreamChannel() {
50    mtype m, p;
51
52  end:
53    do
54    :: atomic { toAnalyzer?[m, p] && nfull(analyzer) ->
55        toAnalyzer?m, p;
56        if
57        :: analyzer!m, p
58        :: analyzer!m, WrongParameter
```


Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

```
59        :: analyzer!WrongMessage, WrongParameter
60        fi
61      }
62    od
63  }
64
65  // Remove all pending messages from all channels
66  inline emptyChannels() {
67    atomic {
68      do
69      :: nempty(toSystem) -> toSystem?_, _
70      :: nempty(system) -> system?_, _
71      :: nempty(systemCtrl) -> systemCtrl?_, _
72      :: nempty(toAnalyzer) -> toAnalyzer?_, _
73      :: nempty(analyzer) -> analyzer?_, _
74      :: nempty(analyzerCtrl) -> analyzerCtrl?_, _
75      :: timeout -> break
76      od
77    }
78  }
79
80  // Analyzer process
81  active proctype analyzerProc() {
82    mtype m, p;
83    bool systemFinished = false;
84
85    do
86    :: 1 -> break
87    :: 1 ->
88
89  progress:
90      run systemProc();
91
92      // Negotiation phase ////////////////////////
93
94      if
95      :: oneTimeNegotiation && negotiationDone -> skip
96      :: else ->
97
98        if
99        :: analyzer?m, p ->
100         if
101         :: m == FeaturesDeclaration ->
102           if
103           :: p == OneTimeNegotiation ->
104             oneTimeNegotiation = true
105           :: p == OneTimeSetup ->
106             oneTimeSetup = true
107           :: p == SkipVariablesDeclaration ->
```


Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

```
108             skipVariablesDeclaration = true
109           :: p == OTN_OTS ->
110             oneTimeNegotiation = true;
111             oneTimeSetup = true
112           :: p == OTN_SVD ->
113             oneTimeNegotiation = true;
114             skipVariablesDeclaration = true
115           :: p == OTN_OTS_SVD ->
116             oneTimeNegotiation = true;
117             oneTimeSetup = true;
118             skipVariablesDeclaration = true
119           :: p == OTS_SVD ->
120             oneTimeSetup = true;
121             skipVariablesDeclaration = true
122           :: else -> skip
123           fi
124         :: else ->
125           toSystem!FeaturesNegotiation, Cancel;
126           goto endAnalyzer
127         fi
128       :: timeout -> goto endAnalyzer
129       fi;
130 
131       if
132       :: toSystem!FeaturesNegotiation, Ok
133       :: toSystem!FeaturesNegotiation, Cancel -> goto endAnalyzer
134       fi
135 
136     fi;
137 
138     // Setup phase ///////////////////////////////
139 
140     if
141     :: oneTimeSetup && setupDone -> skip
142     :: else ->
143 
144       if
145       :: skipVariablesDeclaration -> break
146       :: else ->
147 
148         if
149         :: analyzer?m, p ->
150           if
151           :: m == VariablesDeclaration
152           :: else -> goto endAnalyzer
153           fi
154         :: timeout -> goto endAnalyzer
155         fi
156 
```


Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

```
157       fi;
158
159       toSystem!VariableSelection, NoneAny;
160
161       toSystem!ParameterSetup, NoneAny;
162
163     fi;
164
165     if
166     :: analyzer?m, p->
167       if
168       :: m == Report && p == Ready
169       :: else -> goto endAnalyzer
170       fi
171     :: timeout -> goto endAnalyzer
172     fi;
173
174     // Execution phase //////////////////////////
175
176     toSystem!Command, Start;
177
178     if
179     :: analyzer?m, p ->
180       if
181       :: m == Report && p == Started
182       :: else -> goto endAnalyzer
183       fi
184     :: timeout -> goto endAnalyzer
185     fi;
186
187     systemFinished = false;
188
189     do
190     // Control message from the system
191     :: analyzerCtrl?[m, p] ->
192       analyzerCtrl?m, p;
193       if
194       :: m == Report && (p == Finished || p == Error) ->
195         systemFinished = true
196       :: else -> goto endAnalyzer
197       fi
198     :: else ->
199       if
200       // Message from system
201       :: analyzer?m, p ->
202         if
203         :: m == Report && p == Measure
204         :: else -> goto endAnalyzer
205         fi
```


Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

```
206       // Once system has finished, trace can be consumed at any point
207       :: systemFinished -> break
208       // Result reached
209       :: systemCtrl!Command, Stop ->
210         if
211         :: analyzer?m, p ->
212           if
213           :: m == Report && p == Stopped -> break
214           :: else -> goto endAnalyzer
215           fi
216         :: timeout -> goto endAnalyzer
217         fi
218       :: timeout -> goto endAnalyzer
219       fi
220     od;
221 
222 nextAnalyzer:
223     emptyChannels();
224 
225   od;
226 
227 endAnalyzer:
228   skip
229 }
230 
231 // System process
232 proctype systemProc() {
233   mtype m, p;
234 
235   // Negotiation phase ////////////////////////
236 
237   if
238   :: oneTimeNegotiation -> skip
239   :: else ->
240 
241     if
242     // Don't declare new features after first negotiation
243     :: negotiationDone -> toAnalyzer!FeaturesDeclaration, NoneAny
244     :: else ->
245       if
246       :: toAnalyzer!FeaturesDeclaration, NoneAny
247       :: toAnalyzer!FeaturesDeclaration, OneTimeNegotiation
248       :: toAnalyzer!FeaturesDeclaration, OneTimeSetup
249       :: toAnalyzer!FeaturesDeclaration, SkipVariablesDeclaration
250       :: toAnalyzer!FeaturesDeclaration, OTN_OTS
251       :: toAnalyzer!FeaturesDeclaration, OTN_SVD
252       :: toAnalyzer!FeaturesDeclaration, OTN_OTS_SVD
253       :: toAnalyzer!FeaturesDeclaration, OTS_SVD
254       fi
```


Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

```
255     fi;
256
257     if
258     :: system?m, p ->
259       if
260       :: m == FeaturesNegotiation && p == Ok
261       :: else -> goto endSystem
262       fi
263     :: timeout -> goto endSystem
264     fi;
265
266     negotiationDone = true
267
268   fi;
269
270   // Setup phase //////////////////////////////
271
272   if
273   :: oneTimeSetup && setupDone -> skip
274   :: else ->
275
276     if
277     :: skipVariablesDeclaration -> skip
278     :: else -> toAnalyzer!VariablesDeclaration, NoneAny
279     fi;
280
281     if
282     :: system?m, p ->
283       if
284       :: m == VariableSelection
285       :: else -> toAnalyzer!Report, Error; goto endSystem
286       fi
287     :: timeout -> goto endSystem
288     fi;
289
290     if
291     :: system?m, p ->
292       if
293       :: m == ParameterSetup
294       :: else -> toAnalyzer!Report, Error; goto endSystem
295       fi
296     :: timeout -> goto endSystem
297     fi;
298
299     setupDone = true
300
301   fi;
302
303   toAnalyzer!Report, Ready;
```


Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

```
304
305   // Execution phase ////////////////////////////////
306
307   if
308   :: system?m, p ->
309     if
310     :: m == Command && p == Start
311     :: else -> goto endSystem
312     fi
313   :: timeout -> goto endSystem
314   fi;
315
316   toAnalyzer!Report, Started;
317
318   do
319   // Message from analyzer, check before
320   :: systemCtrl?[m, p] ->
321     systemCtrl?m, p;
322     if
323     :: m == Command && p == Stop ->
324       if
325       :: toAnalyzer!Report, Stopped
326       :: timeout
327       fi;
328       break
329     :: else -> skip
330     fi
331   :: else ->
332     if
333     // Full queue = message cannot be delivered => quit
334     :: full(analyzer) -> analyzerCtrl!Report, Error; break
335     :: nfull(analyzer) ->
336       if
337       // New Measure report
338       :: toAnalyzer!Report, Measure
339       // System finished
340       :: analyzerCtrl!Report, Finished -> break
341       fi
342     fi
343   od;
344
345 endSystem:
346   skip
347 }
```

Listing B.1: Promela model of communication protocol

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

B.2 Protocol buffer message definitions

The messages of the communication protocol between Spin and the system under analysis has been defined using Google Protocol Buffers [14]. Each message is composed of a series of fields, which can be either required, optional or repeated (i.e. can appear any number of times). Fields can be of any of the basic types, such as integers and floating point numbers of different sizes, or other messages. Each field is associated with a numeric tag, used when serializing and deserializing messages to identify the fields. As long as new required fields are not added, messsage definitions can be extended and be decoded by peers that are not aware of these extensions.

Since that serialized protocol buffer messages are not self-delimited, and additional, fixed-size message type `Header` was defined to include the length of the following message, as well as its type. The latter was not strictly necessary due to the way the protocol is defined, but was included nonetheless for debugging and future-proofing purposes. This allows several messages to be sent over a stream, e.g. a TCP socket, which can be separated and decoded easily by the receptor.

Listing B.2 contains the full definition of the protocol messages. The purpose of most messages has already been explained in Section 3.2.

```
1  package optysim;
2
3  option java_package = "es.uma.lcc.salmeron.optysim";
4  option java_outer_classname = "OptySimProto";
5
6  // Accepted types of variables.
7  enum VariableType {
8    INT = 0;
9    LONG = 1;
10   FLOAT = 2;
11   DOUBLE = 3;
12   STRING = 4;
13 }
14
15 // Type of messages declared here, excluding Header. Values should
       be updated
16 // when a new message is included, but not when one is deleted.
17 enum MessageType {
18   UNKNOWN = 0;
19   VARIABLE_VALUE = 1;
20   VARIABLE_SETUP = 2;
21   PARAMETER_SETUP = 3;
22   COMMAND = 4;
23   REPORT = 5;
24   VARIABLE_SELECTION = 6;
25 }
26
```


Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

```
27 // Header preceding another message, which includes the type and
      length of said
28 // following message. This message is always 4 byte long, to ensure
      easy
29 // reception and decoding of messages in a stream.
30 message Header {
31   // We use fixed32 instead of MessageType to ensure fixed encoding
        size
32   required fixed32 type = 1;
33   required fixed32 length = 2;
34 }
35
36 // A variable identifier and value.
37 message VariableValue {
38   required int32 id = 1;
39   optional VariableType type = 2;
40   optional int32 intValue = 3;
41   optional int64 longValue = 4;
42   optional float floatValue = 5;
43   optional double doubleValue = 6;
44   optional string stringValue = 7;
45 }
46
47 // Initial configuration of variables, i.e. measures vs. parameters,
      types.
48 message VariableSetup {
49   message VariableDeclaration {
50     required int32 id = 1;
51     optional string name = 2;
52     optional VariableType type = 3 [default = INT];
53     optional bool parameter = 4 [default = false];
54   }
55
56   repeated VariableDeclaration variable = 1;
57 }
58
59 // Initial values for parameters.
60 message ParameterSetup {
61   repeated VariableValue parameter = 1;
62 }
63
64 // Command sent from the analysis module to the SUT.
65 message Command {
66   enum CommandType {
67     START = 0; // Start the SUT
68     STOP = 1; // Stop the SUT
69     RESET = 2; // Reset the SUT, including parameters and other
          settings
70     REPORT = 3; // Report the most recent values of the measures
```


Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

```
71      OTHER = 4; // Another command, check string parameter
72    }
73
74    optional CommandType type = 1 [default = OTHER];
75    optional string command = 2;
76  }
77
78  // Report from the SUT to the analysis module.
79  message Report {
80    enum ReportType {
81      READY = 0; // The SUT is ready to be run
82      STARTED = 1; // The SUT has started
83      FINISHED = 2; // The SUT has finished normally
84      ERROR = 3; // The SUT has encountered an error
85      MEASURE = 4; // The SUT reports new values for some of the
           measures
86      STOPPED = 5; // The SUT has stopped after receiving a STOP
           command
87      RESET = 6; // The SUT has reset after receiving a RESET command
88    }
89
90    required ReportType type = 1;
91    optional string msg = 2;
92    repeated VariableValue measure = 3;
93  }
94
95  // A subset of selected variables, possibly with a remapping of
      their ids.
96  message VariableSelection {
97    message SelectedVariable {
98      required int32 id = 1;
99      optional int32 newId = 2;
100   }
101
102   repeated SelectedVariable variable = 1;
103 }
```

Listing B.2: Protocol buffers definitions for communication protocol

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

Appendix C: Improved E-model algorithm

Listings C.1 and C.2 show the improved state transition counting algorithm mentioned in section 5.4.2. Listing C.1 shows the part of the algorithm that reacts to a packet reception, while C.2 shows the reaction to a packet loss. In both cases, the `time` parameter is the time in which the current packet was either received or lost.

Publicaciones y Divulgación Científica
uma UNIVERSIDAD DE MÁLAGA

```
recv (time) {
    pkt++;
    if (start == -1)
        start = time;
    endUncert = time;

    if (state == GAP) {
        lost = 0;
        c11++;
        stateRecv++;
        endCert = time;
    }
    else if (state == FIRST_LOSS) {
        state = GAP_RECOVERY;
    }
    else if (state == GAP_RECOVERY) {
        if (pkt == gmin) {
            state = GAP;
            lost = 0;
            c11 += pkt - 1;
            c41++;
            c14++;
            stateRecv += pkt;
            stateLost++;
            endCert = time;
        }
    }
    else if (state == BURST) {
        state = BURST_RECOVERY;
        startUncert = time;
    }
    else if (state == BURST_RECOVERY) {
        if (pkt == gmin) {
            state = GAP;
            lost = 0;
            c11 += pkt - 1;
            c31++;
            start = startUncert;
            endCert = time;
        }
    }
}
```

Listing C.1: Improved state transition counting algorithm; packet reception event

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

```
lost (time) {
    lost++;
    if (start == -1)
        start = time;
    endUncert = time;

    if (state == GAP) {
        state = FIRST_LOSS;
        startUncert = time;
    }
    else if (state == FIRST_LOSS) {
        state = BURST;
        c13++;
        c33++;
        stateRecv = 0;
        stateLost = 2;
        start = startUncert;
        endCert = time;
    }
    else if (state == GAP_RECOVERY) {
        state = BURST;
        c13++;
        c32++;
        c23++;
        c22 += pkt - 1;
        stateRecv = pkt;
        stateLost = 2;
        start = startUncert;
        endCert = time;
    }
    else if (state == BURST) {
        c33++;
        stateLost++;
        endCert = time;
    }
    else if (state == BURST_RECOVERY) {
        state = BURST;
        c32++;
        c23++;
        c22 += pkt - 1;
        stateRecv += pkt;
        stateLost++;
        endCert = time;
    }

    pkt = 0;
}
```

Listing C.2: Improved state transition counting algorithm; packet loss event

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

Appendix D: Resumen en español

Los sistemas *software* y *hardware* se encuentran cada vez más presentes en nuestras vidas, en multitud de campos de aplicación y de cualquier tamaño. El análisis de estos sistemas es una tarea dura pero necesaria para garantizar que cumplan con sus requisitos. Estos requisitos pueden ser de varios tipos, como evitar comportamientos erróneos u ofrecer un rendimiento satisfactorio.

Existen muchas técnicas y herramientas diseñadas para atacar este problema. Por lo general, se aplican distintas técnicas dependiendo del tipo de sistema, fase de desarrollo o tipo de análisis. Por ejemplo, la estimación del rendimiento de un sistema de red se podría hacer analíticamente durante el planteamiento, mediante simuladores de redes en el desarrollo, y finalmente con herramientas de medición durante el despliegue.

El *model checking* [45] es una de estas técnicas de análisis, que ha sido aplicada con éxito tanto en sistemas *software* como *hardware*. Las herramientas que usan esta técnica, como Spin [80], son conocidas como *model checkers*. Un *model checker* analiza el espacio de estados de un sistema, es decir, todos sus posibles caminos de ejecución, para comprobar si el sistema cumple una propiedad dada. En caso negativo, el *model checker* devuelve un camino de ejecución que conduce a un error a modo de contraejemplo.

El *model checking* se ha aplicado tradicionalmente a modelos de sistemas. Por ejemplo, Spin analiza modelos escritos en el lenguaje de especificación Promela. No obstante, el *model checking* también se ha aplicado a sistemas reales, como pueden ser programas Java o C [74][31]. Sin embargo, los sistemas reales suelen revelar uno de los problemas más conocidos del *model checking*: la explosión del espacio de estados. Según aumenta la complejidad del sistema a analizar, su espacio de estados crece rápidamente, hasta llegar a un punto en el que no es factible analizarlo. Existen técnicas dedicadas a mitigar este problema, como las técnicas de abstracción, que buscan reducir el tamaño del espacio de estados pero conservando la capacidad de analizar las propiedades deseadas.

Otra de las técnicas más extendidas para el análisis de *software* durante su desarrollo es el *testing* [28]. Algunas técnicas de *testing* tratan el sistema como una “caja negra”, observando sólo la respuesta del sistema a un estímulo de entrada, mientras que otras pueden acceder al comportamiento interno del mismo.

Todas estas técnicas tienen sus ventajas e inconvenientes. Además, muchas de ellas están orientadas a un tipo de sistema o análisis en concreto. Si un usuario quisiera realizar

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

un análisis completo de un sistema necesitaría aprender a manejar varias de ellas, cada una probablemente con formas diferentes de expresar los requisitos del sistema, o incluso con diferentes modelos del sistema.

D.1 El marco de trabajo de análisis OptySim

En muchos casos, analizar el espacio de estados de un sistema al completo puede no resultar práctico por su tamaño, y en otros directamente irrealizable por no encontrarse disponible. En su lugar, OptySim trata con un conjunto de trazas de ejecución, que representan un subconjunto del espacio de estados completo del sistema. Para obtener dichas trazas el sistema se ejecuta repetidas veces, posiblemente variando parámetros del sistema de acuerdo a las instrucciones del usuario, generándose una traza por cada ejecución. El contenido de las trazas depende de cada sistema, y además puede variar dependiendo de las necesidades del análisis. Para ello se pueden aplicar una de las *proyecciones* que se han definido, y que transforman trazas completas en trazas abstractas con una menor, pero suficiente para los propósitos del análisis, cantidad de información.

El análisis está guiado por uno o más objetivos establecidos por el usuario, y que le dan al análisis el significado pretendido por el usuario. Es posible escribir objetivos de diferentes formas, como simples asertos o fórmulas de lógica temporal lineal (LTL) [102]. Los objetivos pueden indicar tanto propiedades deseables del sistema, por ejemplo una meta de rendimiento, como propiedades que no deben ocurrir, por ejemplo una condición de error.

En muchos tipos de análisis se busca un sólo resultado por traza, en lugar de encontrar todos aquellos puntos en los que algún objetivo se da (tal y como sería usual en el *model checking*). En dicho caso, el análisis de la traza se detiene en cuanto se ha obtenido una respuesta, y se continúa el análisis por la siguiente traza disponible. Dado que las trazas se analizan a la vez que son generadas por el sistema, en lugar de una vez se ha obtenido la traza al completo, esto puede conducir a ahorros significativos de tiempo y recursos tanto en el análisis como en las ejecuciones en sí.

D.1.1 Visión general del análisis

La Figura D.1 muestra una visión general de los elementos y la operación de OptySim, divididos en cuatro fases. En primer lugar, en la fase de preparación, el usuario debe proporcionar tanto el sistema que se desea analizar, como una descripción del análisis que se quiere realizar. Esta descripción incluye la declaración del espacio de parámetros que se quiere explorar, las variables de interés del sistema, y los objetivos que se quieren comprobar sobre las trazas de ejecución generadas.

En la segunda fase, generación de código, la descripción del análisis se procesa junto con una plantilla para obtener una especificación Promela que contiene el algoritmo

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

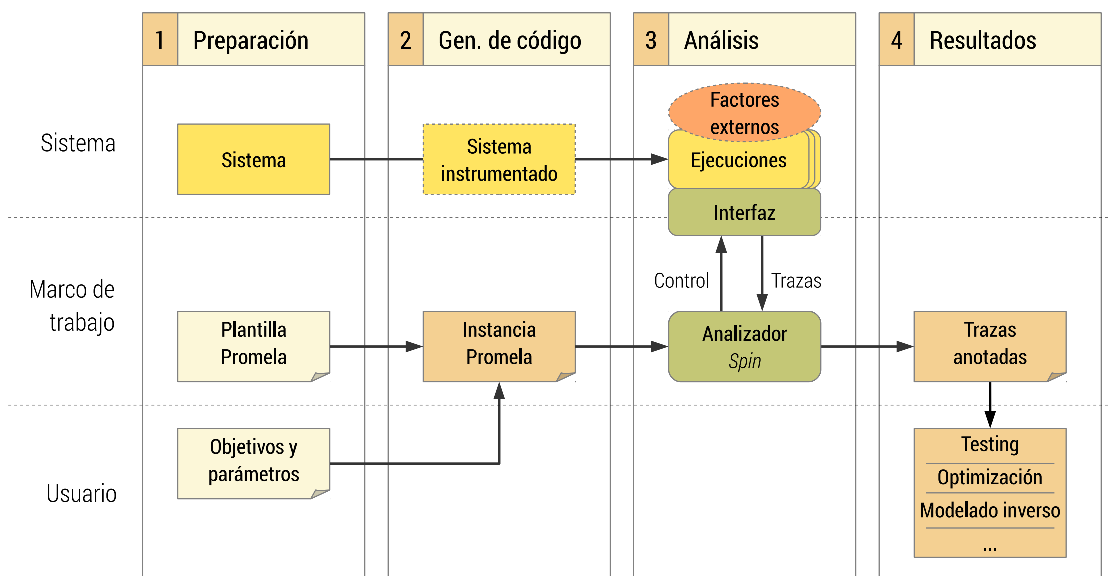


Figura D.1: El marco de trabajo de análisis OptySim

de análisis y la capacidad de comunicarse remotamente con el sistema. Mediante esta especificación y el *model checker* Spin se genera un programa ejecutable que contiene el analizador.

El análisis en sí se realiza en el siguiente paso. Por un lado, se explora el espacio de parámetros definido por el usuario. Por otro lado, cada combinación de parámetros completa da lugar a una ejecución del sistema aplicando dichos parámetros, generando una traza de ejecución para analizar. El lanzamiento del sistema, la configuración de los parámetros y la transmisión de la traza durante la ejecución se realizan a través de un protocolo de comunicación definido para la ocasión.

Las trazas de ejecución son transmitidas al analizador mientras el sistema se ejecuta. El analizador implementado con Spin reconstruye estas trazas y las incorpora como si fueran parte de la especificación Promela que está explorando. Para ello, el estado global del analizador incluye las variables que componen cada estado de la traza de ejecución, entre otras variables. La reconstrucción de la traza de ejecución se realiza en un bucle interno del analizador. En este bucle se reciben nuevos estados de la traza de ejecución (mediante el uso de código C empotrado en Promela), que son convertidos en nuevos estados Spin actualizando las variables globales correspondientes (ver Figura D.2).

El protocolo de comunicación entre el analizador y el sistema está compuesto por varias fases. En primer lugar el sistema declara sus características y puede requerir ciertas capacidades al analizador. Si esta negociación es exitosa, a continuación se declaran las variables disponibles, de las cuales el analizador selecciona un subconjunto, y se proporcionan los parámetros de ejecución. Por último, el sistema se ejecuta y transmite

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

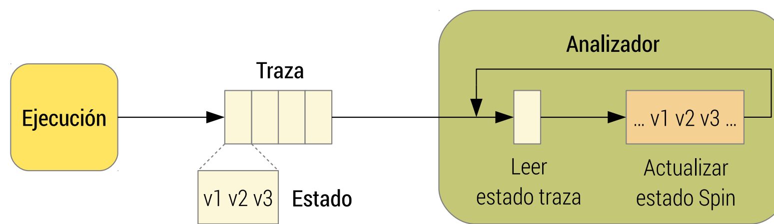


Figura D.2: Reconstrucción de trazas de ejecución en Spin

de forma continua los estados relevantes de la traza de ejecución al analizador. Este protocolo se ha definido mediante *Protocol Buffers* [14], y ha sido analizado con Spin.

Aunque las trazas de ejecución son lineales por naturaleza, la exploración que realice Spin de ellas no tiene por que serlo si en el análisis se incluye un objetivo escrito como fórmula LTL o similar. Spin traduce las fórmulas LTL a procesos Promela que se ejecutan de forma sincronizada con el resto del sistema. Estos procesos, conocidos como *never claim*, pueden contener indeterminismo, lo que provoca que la exploración de la traza sea también indeterminista. Para tratar con este indeterminismo, los estados recibidos del sistema se guardan en una pila de la que son extraídos bajo demanda. En el caso de que se realice una vuelta atrás durante la exploración, se comenzará extrayendo los estados almacenados de esta manera en lugar de recibir estados nuevos.

Por último, las trazas son devueltas al usuario, indicando qué objetivo ha cumplido cada una.

D.1.2 Objetivos

Los objetivos dotan de sentido al análisis, y permiten reducir los recursos y el tiempo requeridos. Los objetivos se pueden escribir usando distintos formalismos, siempre que se puedan traducir a autómatas *never claim* aceptados por Spin, como es el caso de las fórmulas LTL. Además, OptySim soporta la definición de asertos sobre estados, que son comprobados sin usar el mecanismo correspondiente de Spin. Como limitación, sólo es posible establecer un objetivo en forma de *never claim* a la vez.

Los objetivos se etiquetan además para distinguir si su significado es positivo o negativo. Para ello contendrán *accept* o *reject* como parte del nombre, respectivamente. Es conveniente mencionar que la interpretación que Spin hace de los objetivos no depende de esta etiqueta. Cuando un objetivo se cumple en una traza, en analizador anota dicha traza con la etiqueta correspondiente. Si se han definido varios objetivos, el primero en cumplirse determinará la etiqueta que se le dará a la traza.

Es usual definir varios objetivos para limitar el análisis y podar las trazas improductivas lo antes posible. Las siguientes fórmulas sirven a modo de ejemplo, definiendo una fórmula LTL como objetivo de tipo *accept*, y dos asertos etiquetados con *reject* que definen condiciones indeseadas:

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

$$\begin{aligned} ltlAccept &: \ \Diamond(var = 1 \ \land \ \Diamond(var = 2)) \\ reject &: \ (var < 0 \ \lor \ var > 2) \\ reject &: \ time > 1000 \end{aligned} \quad \text{(D.1)}$$

Por último, es posible añadir restricciones adicionales sobre las configuraciones de parámetros que pueden ser generadas, y relaciones entre las configuraciones de parámetros que permitan inferir resultados sin tener que llegar a ejecutar el sistema.

D.2 Proyección de trazas de ejecución

Para poder reducir el tamaño de las trazas de ejecución a analizar, y de los estados que componen las trazas, se han definido varias proyecciones. Decimos que una traza completa se *proyecta*, obteniendo una traza abstracta con menor cantidad de información. Dicha traza debería ser, no obstante, suficiente para los propósitos de un análisis concreto.

En primer lugar, definimos una proyección básica ρ_V en la que el contenido de cada estado se ve reducido, de manera que sólo se conservan un subconjunto V de variables del sistema. Esta proyección se puede considerar similar a la técnica del *cono de influencia* [45] y la ocultación selectiva de datos de Spin [80]. Si las variables necesarias para evaluar una fórmula están contenidas en V, entonces esta proyección no altera la evaluación de la fórmula.

Sin embargo, el algoritmo de búsqueda en profundidad doble usado por Spin puede llevar a conclusiones erróneas si la traza abstracta contiene estados abstractos repetidos. Estos pueden llevar a Spin a deducir la existencia de un ciclo que no está presente en la traza original. Para solventar este problema definimos dos proyecciones adicionales.

La *proyección contador*, ρ_V^c, añade una variable fresca adicional denominada *contador* a los estados. Dicha variable se incrementa de forma monotónica para cada nuevo estado. De esta forma se garantiza que Spin no encontrará ciclos espurios, a costa de que realmente no encontrará *ningún* ciclo en la traza abstracta. Esta proyección es fácil de aplicar y se puede emplear para aquellas fórmulas que no requieran de detección de ciclos.

Dada una función *hash* h, la *proyección hash*, ρ_V^h, añade una variable fresca adicional a cada estado. Dicha variable contendrá el resultado de aplicar la función h al estado completo. Esta variable garantiza que dos estados iguales tendrán la misma representación abstracta, mientras que dos estados distintos no la tendrán, con el grado de precisión permitido por la función h. Esta proyección es más costosa de implementar computacionalmente, pero permite emplear fórmulas que requieran de detección de ciclos.

Por último, definimos unas versiones refinadas de las dos proyecciones anteriores, en las que se intenta reducir el número de estados en la traza abstracta. Las *proyecciones*

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

contador y hash plegadas, ϕ_V^c y ϕ_V^h respectivamente, sólo proyectan en la traza abstracta aquellos estados en los que se hayan modificado las variables presentes en V. Llamamos *plegados* a los estados que no han sido proyectados. Como contrapartida, un bucle infinito que ocurra dentro de los estados plegados no llegaría a ser detectado nunca y el análisis no llegaría a terminar. Para ello, definimos la *proyección hash plegada limitada*, $\phi_{V,l}^h$, donde el número de estados plegados consecutivos está limitado a l. Al alcanzarse este límite, todos los estados siguientes serán proyectados, hasta llegar a un estado que hubiera sido proyectado de forma habitual, reestableciéndose este límite.

D.3 Integración y casos de estudio: ns-2

Para el primer grupo de casos de estudio, se ha integrado el simulador de redes ns-2 [11] con OptySim. Los casos de estudio tratan con el análisis de fiabilidad, rendimiento y optimización de un escenario de transmisión de vídeo sobre TCP, validación de modelos de calidad de experiencia (QoE) para llamadas de voz sobre IP (VoIP), y ajuste de modelos de indicadores clave de rendimiento (KPIs) basados en pruebas de campo para su uso en simulaciones.

ns-2 es uno de los simuladores de redes más usados en el mundo académico en los últimos años. Este simulador está implementado con una mezcla de C++ y OTcl. Su núcleo consiste en un simulador de eventos discretos, donde los eventos son encolados y atendidos de acuerdo a su tiempo de disparo. Para la extracción de trazas de ns-2, consideramos que cada evento da lugar a un nuevo estado.

Se ha implementado un componente en C++ y OTcl para integrar ns-2 con OptySim. Las principales responsabilidades de este componente son implementar el protocolo de comunicación entre el simulador y el analizador, y extraer las trazas de ejecución según la proyección elegida por el usuario. Este componente sólo soporta la proyección contador descrita anteriormente. Dicho componente se ha implementado en dos partes: una genérica en C++, y otra específica para ns-2. Esta separación permitirá integrar otros sistemas que soporten C++ con menor esfuerzo.

D.3.1 Caso de estudio: vídeo sobre TCP

Para el primer caso de estudio se ha definido un escenario de transmisión de vídeo sobre TCP hacia un teléfono móvil. Una incidencia habitual en este tipo de entornos móviles son las desconexiones debidas al entorno, por ejemplo por mala recepción de la señal móvil o por cambio entre celdas. El protocolo TCP no fue desarrollado con este tipo de desconexiones en mente, y su reacción ante ellas no es adecuada. Por ello han surgido varias alternativas que intentan paliar estas deficiencias. Es el caso de FreezeTCP [69], una variante de TCP que mejora la reacción ante las desconexiones predecibles.

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

En primer lugar se realizó un análisis de fiabilidad tanto del cliente y servidor de vídeos implementados para la ocasión, como de la implementación de FreezeTCP usada. Los objetivos usados llevaron a descubrir un pequeño fallo en la implementación de este último, así como otro en el módulo de TCP de ns-2.

A continuación se estudió el rendimiento de la reproducción de vídeo en presencia de desconexiones, la influencia del *buffer* de reproducción y la efectividad de FreezeTCP en diversas condiciones. En particular, los objetivos se centraban en evitar que el *buffer* de reproducción se vaciase durante la transmisión. Gracias a este análisis se obtuvieron una serie de configuraciones de parámetros que proporcionaban un rendimiento satisfactorio de acuerdo a los objetivos. Por último, se construyó una versión del cliente que adaptaba sus parámetros tras cada desconexión, en el caso de que las condiciones del entorno hubieran cambiado, empleando las configuraciones obtenidas en el análisis anterior. Esta versión mejoraba a aquellas que empleaban una configuración estática cuando ocurrían cambios en las características del entorno móvil tras las reconexiones.

D.3.2 Caso de estudio: validación de E-model

El siguiente caso de estudio trata sobre la validación de modelos. Este caso de estudio se centró en modelos de QoE diseñados para VoIP y, en particular, en el modelo conocido como E-model [10][1]. Este modelo busca aproximar la valoración subjetiva de los usuarios mediante el uso de datos de la transmisión como tiempos de retraso o tasas de paquetes perdidos. Una extensión propuesta por Clark [39] mejora esta aproximación cuando en la llamada se producen pérdidas en ráfaga.

Previamente al análisis, se desarrolló una nueva mejora sobre la versión de Clark, cuya principal ventaja era la obtención de resultados de forma más inmediata que el modelo original. Para comprobar que ambos modelos seguían produciendo resultados comparables, se preparó un escenario de validación donde las mismas llamadas VoIP eran analizadas por ambas versiones del E-model simultáneamente. El objetivo de validación se definió en base a características estadísticas observables en los resultados producidos por ambos. Se concluyó que ambos modelos producían resultados similares en presencia de errores aleatorios, pero se separaban sensiblemente según el patrón de las ráfagas de error, debido a la mayor sensibilidad de nuestra solución ante éstas.

D.3.3 Caso de estudio: modelado inverso de KPIs

El último caso de estudio con ns-2 trata sobre el modelado inverso de indicadores clave de rendimiento (KPIs) a partir de pruebas de campo. El uso de simulaciones durante el desarrollo y despliegue de sistemas en red es una forma conveniente y barata de estudiar su comportamiento, frente al coste que suponen las pruebas de campo. No obstante, es difícil capturar las características de los entornos reales en escenarios simulados. Los KPIs son medidas de diversa índole que influyen en el rendimiento de un sistema. El

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

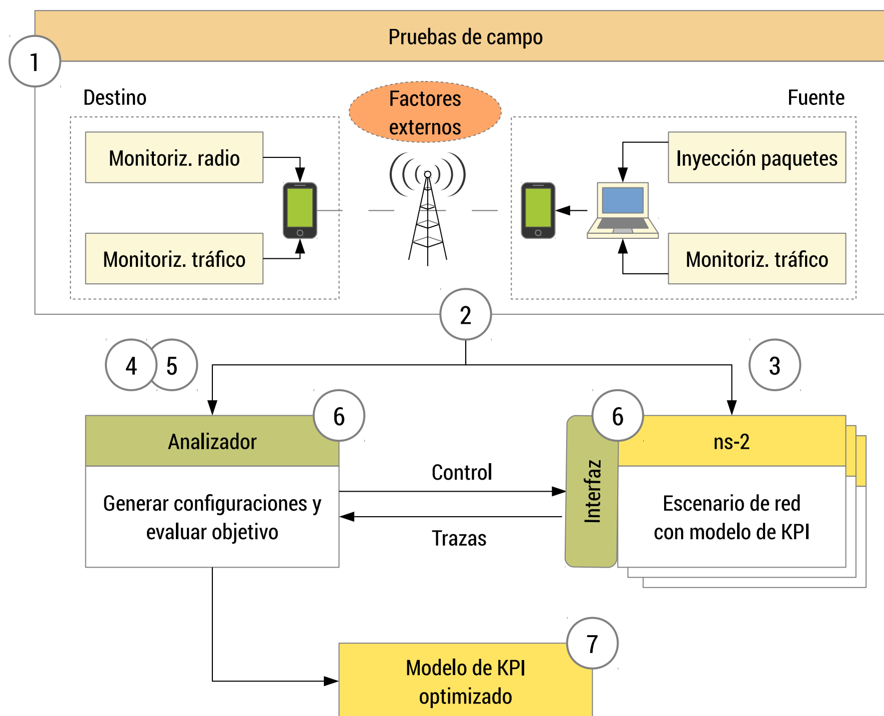


Figura D.3: Metodologia para el modelado inverso de KPIs

uso en simulaciones de KPIs realistas basados en pruebas de campo permite mejorar su fidelidad y efectividad.

Para ello se propone una metodología que hace uso de la herramienta de perfilado para móviles SymPA [53] en combinación con OptySim para configurar modelos de KPIs realistas, resumida en la Figura D.3. En un primer paso se realizan pruebas de campo, extrayendo métricas relevantes para el KPI que se quiere modelar. A continuación se prepara un escenario ns-2 que replica el escenario de las pruebas, donde se incluye un modelo parametrizado para el KPI. Como objetivo para el análisis se propone la similitud de ciertas estadísticas extraídas de las pruebas de campo con la salida generada por el modelo del KPI. Como resultado, se obtendrán aquellas configuraciones de parámetros del KPI que han cumplido dicho objetivo, y que representan de forma apropiada las características del KPI observadas en las pruebas de campo.

Como caso de estudio se propone el modelado del *jitter* en llamadas VoIP. El *jitter* se define como la diferencia entre el tiempo esperado de llegada de un paquete y el tiempo real de llegada. El modelo propuesto de *jitter* se trata de un modelo de error ns-2 colocado en el enlace que representa el medio inalámbrico, y que contiene una distribución de probabilidad que determina el retraso aplicado a los paquetes en dicho enlace. Como resultado se obtuvieron varias configuraciones de parámetros válidas, según la distribución de probabilidad empleada.

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

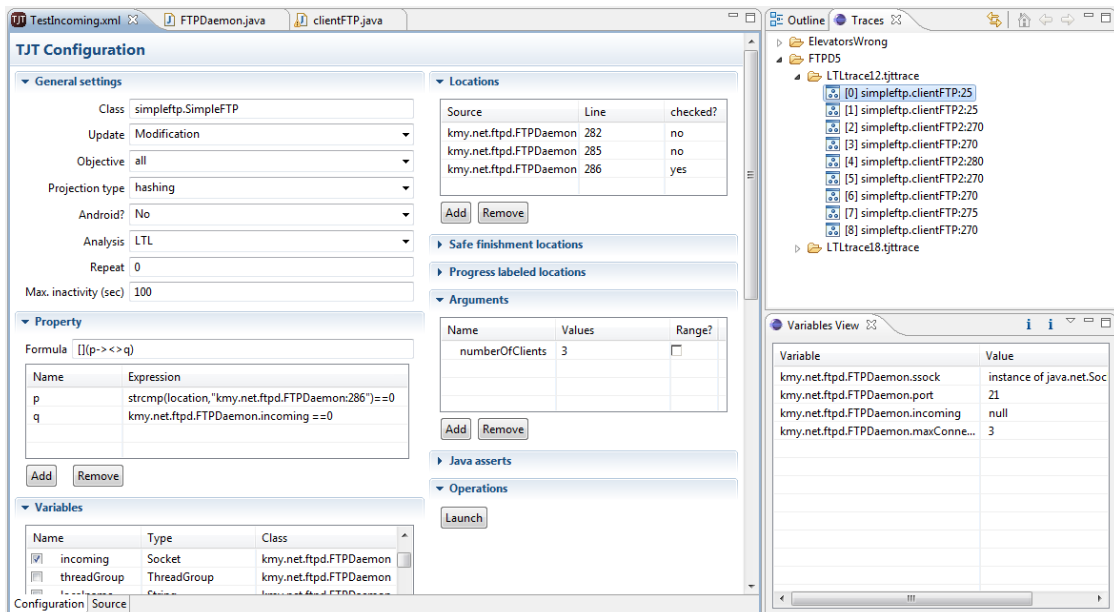


Figura D.4: Captura de pantalla del *plug-in* para Eclipse

D.4 Integración y casos de estudio: Java

Para el segundo grupo de casos de estudio, se ha integrado OptySim con una máquina virtual de Java (JVM) para analizar programas escritos en dicho lenguaje de programación. En esta ocasión, todos los casos de estudio están enfocados a la depuración de programas.

La integración con la JVM para extraer trazas de ejecución se ha realizado a través de la interfaz de depuración de Java (JDI). Gracias a esta interfaz es posible controlar las ejecuciones y extraer trazas de forma transparente a los programas Java. Para esta integración se ha implementado la proyección *hash*, lo que permite el uso de objetivos que requieran de detección de ciclos en las trazas.

Además se ha desarrollado un *plug-in* para el entorno de desarrollo Eclipse, mostrado en la Figura D.4. Este *plug-in* permite definir casos de prueba, incluyendo los objetivos a comprobar y el espacio de parámetros a explorar, y ejecutar el análisis. Una vez concluido, es posible inspeccionar las trazas devueltas desde Eclipse, para encontrar la causa de los errores informados.

Como casos de estudio, se seleccionaron una serie de servidores implementados en Java y de libre disposición: un servidor web, un servidor FTP y otro NFS. Además, se desarrollaron otros ejemplos específicos para comprobar la utilidad de la integración.

Frente a un *model checker* específico para Java como JPF [74], nuestra integración de OptySim ofrece una serie de ventajas e inconvenientes. Por un lado, no se realiza una exploración exhaustiva de los programas, sino sólo de el subconjunto dado por las trazas exploradas. Para programas de gran tamaño, no obstante, esta puede ser una forma

Publicaciones y Divulgación Científica
UMA UNIVERSIDAD DE MÁLAGA

de asegurar un análisis práctico. Por otro lado, OptySim permite definir propiedades en las que se haga mención a variables del programa y puntos de ruptura, entre otros elementos, mientras que la extensión para LTL de JPF [90] sólo permite emplear la entrada a métodos.

D.5 Conclusiones

Las secciones anteriores han resumido el funcionamiento y los usos de OptySim, nuestro enfoque para al análisis de trazas de ejecución mediante *model checking*. Los principales objetivos cumplidos son el uso de un único marco de trabajo para estudiar distintos tipos de sistema (como simulaciones ns-2 y programas Java), y también la realización de distintos tipos de análisis (como rendimiento o fiabilidad).

Las principales contribuciones del trabajo realizado, y las publicaciones donde han sido presentadas, se pueden resumir en:

- Un marco de trabajo para analizar trazas de ejecución usando *model checking* [93][23]. Este marco está basado en el *model checker* Spin y una plantilla Promela que se especializa antes de cada análisis. Entre sus características se encuentran la generación de configuraciones de parámetros y el soporte de vuelta atrás durante la exploración de trazas.

- Un protocolo para interactuar con un sistema externo y extraer trazas de ejecución. Este protocolo permite la separación del analizador y el sistema analizado, y puede ser reusado por otros sistemas mediante el desarrollo de una interfaz apropiada.

- Una extensión para controlar la exploración de espacios de estados compuestos por trazas independientes. El primer resultado disponible para cada traza es devuelto y el análisis procede con la siguiente, reduciendo el espacio de estados a analizar.

- Dos métodos de abstracción para reducir la información requerida en las trazas de ejecución [23][22]. Estos métodos, llamados *proyección contador* y *hash*, minimizan la cantidad de información necesaria por estado. La proyección *hash* permite también la detección de ciclos. El uso de *plegado* reduce el número de estados en cada traza.

- Integración con el simulador de redes ns-2, con aplicaciones en el análisis de fiabilidad y rendimiento [93][92], validación [105], y modelado de KPIs [55].

- Una extensión del E-model de Clark para la estimación de la calidad de llamadas VoIP con mayor inmediatez [105].

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

- Integración con Java y Eclipse para la depuración de programas Java [23][22]. Mediante el uso de la interfaz JDI, las trazas de ejecución pueden ser extraídas sin modificar el programa original. El uso de la proyección *hash* permite el uso de fórmulas LTL que requieran detección de ciclos en la definición de casos de prueba.

Como trabajos futuros se propone la integración de OptySim con más simuladores de redes, como el simulador de redes ns-3 [77], y herramientas de desarrollo dirigidas por modelos, como Rational Rhapsody [17]. Dado que ambos sistemas están desarrollados con C++, en ambos caso se puede aprovechar el trabajo realizado para la integración con ns-2. También se contempla ampliar el soporte de la proyección *hash*, implementada únicamente en la integración con Java.

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

References

[1] ITU Recommendation G.108. Application of the E-model: A planning guide. Technical report, ITU, 1999. 109, 111, 117, 183

[2] ETSI TS 101 329-5. TIPHON Release 3; Technology Compliance Specification; Part 5: Quality of Service (QoS) measurement methodologies. Technical report, ETSI, 2000. 111, 117

[3] RTP: A Transport Protocol for Real-Time Applications. RFC 3550 (Standard), May 2003. 98

[4] JSR 163: Java™ Platform Profiling Architecture. `https://www.jcp.org/en/jsr/detail?id=163`, September 2004. 132

[5] Technical Specification Group Radio Access Network; UE Radio Access capabilities (Release 5). RFC 3550 (Standard), May 2006. 98

[6] The MD5 Message-Digest Algorithm. RFC 1321 (Standard), May 2006. 71

[7] XDR: External Data Representation Standard. RFC 4506 (Standard), May 2006. 136

[8] ITU Recommendation X.680. Abstract Syntax Notation One (ASN.1): Specification of basic notation. Technical report, ITU-T, November 2008. 57

[9] Freeze-TCP (ns-2 and Linux implementations). July 2009. 99, 102

[10] ITU Recommendation G.107. The E-model: a computational model for use in transmission planning. Technical report, ITU, 2009. 109, 110, 111, 183

[11] The Network Simulator - ns-2. August 2010. 24, 87, 182

[12] Apache Velocity. `http://velocity.apache.org/`, December 2013. 51

[13] JAXB Reference Implementation. `https://jaxb.java.net`, December 2013. 60

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

[14] Protocol Buffers: Google's Data Interchange Format. https://code.google.com/p/protobuf/, November 2013. 40, 57, 169, 180

[15] Eclipse - The Eclipse Foundation open source community website. http://www.eclipse.org/, January 2014. 133

[16] IBM - Rational SDL Suite. http://www-01.ibm.com/software/awdtools/sdlsuite/, March 2014. 149

[17] IBM - Software - Rational Rhapsody family. http://www-01.ibm.com/software/awdtools/rhapsody/, March 2014. 149, 187

[18] Java™ Debug Interface. http://docs.oracle.com/javase/7/docs/jdk/api/jpda/jdi/index.html, January 2014. 133

[19] Java™ Debug Wire Protocol. http://docs.oracle.com/javase/7/docs/technotes/guides/jpda/jdwp-spec.html, January 2014. 133

[20] Java™ Platform Debugger Architecture (JPDA). http://docs.oracle.com/javase/7/docs/technotes/guides/jpda/, January 2014. 132

[21] JUnit testing framework. http://www.junit.org, February 2014. 2, 27

[22] Damián Adalid, Alberto Salmerón, María del Mar Gallardo, and Pedro Merino. Testing Temporal Logic on Infinite Java Traces. In *Proceedings of the 10th International Workshop on Modelling, Simulation, Verification and Validation of Enterprise Information Systems (MSVVEIS 2012)*, 2012. 4, 133, 186, 187

[23] Damián Adalid, Alberto Salmerón, María del Mar Gallardo, and Pedro Merino. Using SPIN for automated debugging of infinite executions of Java programs. *Journal of Systems and Software*, **90**:61 – 75, 2014. 4, 133, 186, 187

[24] A. Alvarez, A. Diaz, P. Merino, and F.J. Rivas. Field measurements of mobile services with Android smartphones. In *Consumer Communications and Networking Conference (CCNC), 2012 IEEE*, pages 105–109, 2012. 124

[25] Matteo Maria Andreozzi, Daniele Migliorini, Giovanni Stea, and Carlo Vallati. Ns2Voip++, an enhanced module for VoIP simulations. In *Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques*, SIMUTools '10, 2010. 115

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

[26] MATTEO MARIA ANDREOZZI, GIOVANNI STEA, AND CARLO VALLATI. A framework for large-scale simulations and output result analysis with ns-2. In *Simutools '09: Proceedings of the 2nd International Conference on Simulation Tools and Techniques*, pages 1–7, ICST, Brussels, Belgium, Belgium, 2009. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering). 26, 27

[27] ANDREAS BAUER, MARTIN LEUCKER, AND CHRISTIAN SCHALLHART. Runtime Verification for LTL and TLTL. *ACM Trans. Softw. Eng. Methodol.*, **20**[4]:14:1–14:64, September 2011. 29

[28] BORIS BEIZER. *Software testing techniques*. Dreamtech Press, 2003. 1, 177

[29] DAVE BENSON. protobuf-c: C bindings for Google's Protocol Buffers. `https://code.google.com/p/protobuf-c/`, December 2013. 58

[30] C. BEUST AND H. SULEIMAN. *Next Generation Java Testing: TestNG and Advanced Concepts*. Addison-Wesley Professional, 2007. 2, 28

[31] DIRK BEYER, THOMAS A. HENZINGER, RANJIT JHALA, AND RUPAK MAJUMDAR. The Software Model Checker BLAST: Applications to Software Engineering. *International Journal on Software Tools for Technology Transfer (STTT)*, **9**[5-6]:505–525, 2007. Invited to special issue of selected papers from FASE 2004/05. 1, 11, 177

[32] KARTHIKEYAN BHARGAVAN, CARL A. GUNTER, MOONJOO KIM, INSUP LEE, DAVOR OBRADOVIC, OLEG SOKOLSKY, AND MAHESH VISWANATHAN. Verisim: formal analysis of network simulations. *Software Engineering, IEEE Transactions on*, **28**[2]:129–145, Feb 2002. 25

[33] ERIC BODDEN. A lightweight LTL runtime verification tool for java. In *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, OOPSLA '04, pages 306–307, New York, NY, USA, 2004. ACM. 30

[34] JULIUS R. BÜCHI. On a decision method in restricted second order arithmetic. In ERNEST NAGEL, PATRICK SUPPES, AND ALFRED TARSKI, editors, *Proceedings of the 1960 International Congress on Logic, Methodology and Philosophy of Science (LMPS'60)*, pages 1–11. Stanford University Press, June 1962. 15

[35] L. CARVALHO, E. MOTA, R. AGUIAR, A.F. LIMA, AND J.N. DE SOUZA. An E-model implementation for speech quality evaluation in VoIP systems. In *Computers and Communications, 2005. ISCC 2005. Proceedings. 10th IEEE Symposium on*, pages 933–938, 2005. 110, 113, 114

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

[36] XINJIE CHANG. Network simulations with OPNET. In *WSC '99: Proceedings of the 31st conference on Winter simulation*, pages 307–314, New York, NY, USA, 1999. ACM. 24

[37] CLAUDIO CICCONETTI, ENZO MINGOZZI, AND GIOVANNI STEA. An integrated framework for enabling effective data collection and statistical analysis with ns-2. In *WNS2 '06: Proceeding from the 2006 workshop on ns-2: the IP network simulator*, page 11, New York, NY, USA, 2006. ACM. 26, 115

[38] ALESSANDRO CIMATTI, EDMUND CLARKE, FAUSTO GIUNCHIGLIA, AND MARCO ROVERI. NUSMV: a new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, **2**[4]:410–425, 2000. 11

[39] A. D. CLARK. Modeling the Effects of Burst Packet Loss and Recency on Subjective Voice Quality. In *IPtel 2001 Workshop*, 2001. 87, 109, 111, 114, 183

[40] E. CLARKE, K. MCMILLAN, S. CAMPOS, AND V. HARTONAS-GARMHAUSEN. Symbolic model checking. In RAJEEV ALUR AND THOMASA. HENZINGER, editors, *Computer Aided Verification*, **1102** of *Lecture Notes in Computer Science*, pages 419–422. Springer Berlin Heidelberg, 1996. 11

[41] E. M. CLARKE, E. A. EMERSON, AND A. P. SISTLA. Automatic Verification of Finite-state Concurrent Systems Using Temporal Logic Specifications. *ACM Trans. Program. Lang. Syst.*, **8**[2]:244–263, April 1986. 9

[42] EDMUND M. CLARKE, E. ALLEN EMERSON, AND JOSEPH SIFAKIS. Model Checking: Algorithmic Verification and Debugging. *Commun. ACM*, **52**[11]:74–84, November 2009. 9

[43] EDMUND M. CLARKE AND E.ALLEN EMERSON. Design and synthesis of synchronization skeletons using branching time temporal logic. In DEXTER KOZEN, editor, *Logics of Programs*, **131** of *Lecture Notes in Computer Science*, pages 52–71. Springer Berlin Heidelberg, 1982. 12

[44] EDMUND M. CLARKE, ORNA GRUMBERG, AND KIYOHARU HAMAGUCHI. Another look at ltl model checking. *Formal Methods in System Design*, **10**[1]:47–71, 1997. 13

[45] EDMUND M. CLARKE, ORNA GRUMBERG, AND DORON A. PELED. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999. 1, 10, 11, 18, 69, 177, 181

[46] E.M. CLARKE, O. GRUMBERG, M. MINEA, AND D. PELED. State space reduction using partial order techniques. *International Journal on Software Tools for Technology Transfer*, **2**[3]:279–287, 1999. 67

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

[47] ADRIAN E. CONWAY AND YALI ZHU. A simulation-based methodology and tool for automating the modeling and analysis of voice-over-IP perceptual quality. *Performance Evaluation*, **54**[2]:129–147, 2003. Modelling Techniques and Tools for Computer Performance Evaluation. 26

[48] J.C. CORBETT, M.B. DWYER, J. HATCLIFF, S. LAUBACH, C.S. PASAREANU, ROBBY, AND HONGJUN ZHENG. Bandera: extracting finite-state models from Java source code. In *Software Engineering, 2000. Proceedings of the 2000 International Conference on*, pages 439–448, 2000. 1, 11, 28

[49] C. COURCOUBETIS, M. VARDI, P. WOLPER, AND M. YANNAKAKIS. Memory efficient algorithms for the verification of temporal properties. In EDMUND M. CLARKE AND ROBERTP. KURSHAN, editors, *Computer Aided Verification*, **531** of *Lecture Notes in Computer Science*, pages 233–242. Springer Berlin Heidelberg, 1991. 14, 16

[50] H. DAHMOUNI, A. GIRARD, AND B. SANSO. Analytical jitter model for IP network planning and design. In *Communications and Networking, 2009. ComNet 2009. First International Conference on*, pages 1–7, November 2009. 122

[51] MARCELO D'AMORIM AND KLAUS HAVELUND. Event-based runtime verification of java programs. *SIGSOFT Softw. Eng. Notes*, **30**[4]:1–7, May 2005. 30

[52] MARÍA DEL MAR GALLARDO, PEDRO MERINO, AND ERNESTO PIMENTEL. Comparing under and over-approximations of {LTL} properties for model checking. *Electronic Notes in Theoretical Computer Science*, **76**[0]:131 – 144, 2002. {WFLP} 2002, 11th International Workshop on Functional and (Constraint) Logic Programming, Selected Papers. 18

[53] ALMUDENA DÍAZ, PEDRO MERINO, AND F. JAVIER RIVAS. Mobile Application Profiling for Connected Mobile Devices. *Pervasive Computing, IEEE*, **9**[1]:54–61, jan.-march 2010. 121, 122, 124, 125, 184

[54] ALMUDENA DÍAZ, PEDRO MERINO, AND F. JAVIER RIVAS. QoS analysis of video streaming service in live cellular networks. *Computer Communications*, **33**[3]:322–335, 2010. 124

[55] ALMUDENA DÍAZ, PEDRO MERINO, AND ALBERTO SALMERÓN. Obtaining Models for Realistic Mobile Network Simulations using Real Traces. *Communications Letters, IEEE*, **15**[7]:782–784, 2011. 4, 186

[56] ALMUDENA DÍAZ ZAYAS AND PEDRO MERINO GÓMEZ. SymPa: A Measurement Tool for Evaluating the Performance of IP Services in Mobile Networks. In

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

Proceedings of the 5th ACM Symposium on QoS and Security for Wireless and Mobile Networks, Q2SWinet '09, pages 103–106, New York, NY, USA, 2009. ACM. 124

[57] DORON DRUSINSKY. The Temporal Rover and the ATG Rover. In *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*, pages 323–330, London, UK, UK, 2000. Springer-Verlag. 30

[58] AHMAD FARAJ, XIN YUAN, AND DAVID LOWENTHAL. STAR-MPI: self tuned adaptive routines for MPI collective operations. In *Proceedings of the 20th annual international conference on Supercomputing*, ICS '06, pages 199–208, New York, NY, USA, 2006. ACM. 27

[59] JEAN-CLAUDE FERNANDEZ, CLAUDE JARD, THIERRY JÉRON, AND CÉSAR VIHO. Using on-the-fly verification techniques for the generation of test suites. In RAJEEV ALUR AND THOMASA. HENZINGER, editors, *Computer Aided Verification*, **1102** of *Lecture Notes in Computer Science*, pages 348–359. Springer Berlin Heidelberg, 1996. 16

[60] CHEN FU, ANA MILANOVA, BARBARA GERSHON RYDER, AND DAVID G. WONNACOTT. Robustness Testing of Java Server Applications. *IEEE Transactions on Software Engineering*, **31**[4]:292–311, April 2005. 137

[61] M. M. GALLARDO, P. MERINO, L. PANIZO, AND A. LINARES. A practical use of model checking for synthesis: generating a dam controller for flood management. *Software: Practice and Experience*, **41**[11]:1329–1347, 2011. 10, 19

[62] M. M. GALLARDO, P. MERINO, AND D. SANÁN. Model Checking Dynamic Memory Allocation in Operating Systems. *Journal of Automated Reasoning*, **42**[2]:229–264, 2009. 71

[63] MARÍA DEL MAR GALLARDO, JESÚS MARTÍNEZ, PEDRO MERINO, AND ERNESTO PIMENTEL. aSPIN: a tool for abstract model checking. *Software Tools for Technology Transfer*, **5**[2-3]:165–184, October 2004. 18

[64] MARÍA DEL MAR GALLARDO, PEDRO MERINO, AND ERNESTO PIMENTEL. A generalized semantics of PROMELA FOR ABSTRACT MODEL CHECKING. *Form. Asp. Comput.*, **16**[3]:166–193, 2004. v, 76, 77, 78

[65] ROB GERTH, DORON PELED, MOSHE Y. VARDI, AND PIERRE WOLPER. Simple on-the-fly automatic verification of linear temporal logic. In *Proceedings of the Fifteenth IFIP WG6.1 International Symposium on Protocol Specification, Testing*

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

and Verification XV, pages 3–18, London, UK, UK, 1996. Chapman & Hall, Ltd. 13, 15

[66] DIMITRA GIANNAKOPOULOU AND KLAUS HAVELUND. Automata-Based Verification of Temporal Properties on Running Programs. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering*, ASE '01, pages 412–, Washington, DC, USA, 2001. IEEE Computer Society. 30

[67] PATRICE GODEFROID. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996. 67

[68] PATRICE GODEFROID. Software Model Checking: The VeriSoft Approach. *Formal Methods in System Design*, **26**[2]:77–101, 2005. 25

[69] TOM GOFF, JAMES MORONSKI, D.S. PHATAK, AND VIPUL GUPTA. Freeze-TCP: a true end-to-end TCP enhancement mechanism for mobile environments. In *INFOCOM 2000. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, **3**, pages 1537–1545, Mar 2000. 98, 99, 102, 106, 182

[70] JAMES GOSLING, BILL JOY, JR. GUY L. STEELE, GILAD BRACHA, AND ALEX BUCKLEY. *The Java Language Specification, Java SE 7 Edition*. Addison-Wesley Professional, 1st edition, 2013. 131, 132

[71] G. HAMPEL, G. ABUSH-MAGDER, D. DÍAZ, AND A. DRABECK. The New Paradigm for Wireless Network Optimization: A Synergy of Automated Processes and Human Intervention. *IEEE Communications Magazine*, **43**[3]:13–21, March 2005. 122

[72] K. HAVELUND, M. LOWRY, AND J. PENIX. Formal analysis of a space-craft controller using spin. *Software Engineering, IEEE Transactions on*, **27**[8]:749–765, Aug 2001. 10

[73] K. HAVELUND, A. SKOU, K.G. LARSEN, AND K. LUND. Formal modeling and analysis of an audio/video protocol: an industrial case study using uppaal. In *Real-Time Systems Symposium, 1997. Proceedings., The 18th IEEE*, pages 2–13, Dec 1997. 10

[74] KLAUS HAVELUND AND THOMAS PRESSBURGER. Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer*, **2**[4]:366–381, 2000. 1, 11, 28, 137, 144, 177, 185

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

[75] KLAUS HAVELUND AND GRIGORE ROŞU. An Overview of the Runtime Verification Tool Java PathExplorer. *Form. Methods Syst. Des.*, **24**[2]:189–215, March 2004. 30

[76] J. HEATH, M. KWIATKOWSKA, G. NORMAN, D. PARKER, AND O. TYMCHYSHYN. Probabilistic model checking of complex biological pathways. In C. PRIAMI, editor, *Proc. Computational Methods in Systems Biology (CMSB'06)*, **4210** of *Lecture Notes in Bioinformatics*, pages 32–47. Springer Verlag, 2006. 10

[77] THOMAS R. HENDERSON, SUMIT ROY, SALLY FLOYD, AND GEORGE F. RILEY. Ns-3 project goals. In *Proceeding from the 2006 Workshop on Ns-2: The IP Network Simulator*, WNS2 '06, New York, NY, USA, 2006. ACM. 24, 149, 187

[78] GERARD HOLZMANN AND RAJEEV JOSHI. Model-Driven Software Verification. In SUSANNE GRAF AND LAURENT MOUNIER, editors, *Model Checking Software*, **2989** of *Lecture Notes in Computer Science*, pages 76–91. Springer Berlin / Heidelberg, 2004. 10

[79] GERARD J. HOLZMANN. *Design and validation of computer protocols*. Prentice Hall, 1991. 19, 24

[80] GERARD J. HOLZMANN. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, September 2003. 10, 11, 16, 18, 19, 24, 69, 177, 181

[81] GERARD J. HOLZMANN AND MARGARET H. SMITH. Software Model Checking. In JIANPING WU, SAMUEL T. CHANSON, AND QIANG GAO, editors, *FORTE*, **156** of *IFIP Conference Proceedings*, pages 481–497. Kluwer, 1999. 11

[82] GERARDJ. HOLZMANN. An analysis of bitstate hashing. *Formal Methods in System Design*, **13**[3]:289–307, 1998. 71

[83] PIM KARS. The application of promela and spin in the bos project. In *Proceedings of the Second Spin Workshop*, pages 60–71, Aug 1996. 10, 19

[84] TAKAHIRO KATAGIRI, KENJI KISE, HIROAKI HONDA, AND TOSHITSUGU YUBA. FIBER: A Generalized Framework for Auto-tuning Software. In ALEX VEIDENBAUM, KAZUKI JOE, HIDEHARU AMANO, AND HIDEO AISO, editors, *High Performance Computing*, **2858** of *Lecture Notes in Computer Science*, pages 146–159. Springer Berlin / Heidelberg, 2003. 27

[85] MOONZOO KIM, MAHESH VISWANATHAN, SAMPATH KANNAN, INSUP LEE, AND OLEG SOKOLSKY. Java-MaC: A Run-Time Assurance Approach for Java Programs. *Formal Methods in System Design*, **24**[2]:129–155, 2004. 10.1023/B:FORM.0000017719.43755.7c. 25

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

[86] RALF KREHER. *UMTS Performance Measurement: A Practical Guide to KPIs for the UTRAN Environment*. Wiley, July 2006. 121, 122

[87] RALF KREHER. Key Performance Indicators (KPI) for UMTS and GSM. Technical Report TS 32.410, 3GPP, September 2009. 122

[88] MARTA KWIATKOWSKA, GETHIN NORMAN, AND DAVID PARKER. PRISM: Probabilistic Symbolic Model Checker. In T. FIELD, P. HARRISON, J. BRADLEY, AND U. HARDER, editors, *Proc. 12th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation (TOOLS'02)*, **2324** of *LNCS*, pages 200–204. Springer, 2002. 10, 11

[89] KIM G. LARSEN, PAUL PETTERSSON, AND WANG YI. Uppaal in a nutshell. *International Journal on Software Tools for Technology Transfer*, **1**[1-2]:134–152, 1997. 10

[90] MICHELE LOMBARDI. jpf-ltl – Bitbucket repository. `https://bitbucket.org/michelelombardi/jpf-ltl`, April 2014. 29, 144, 186

[91] A.P. MARKOPOULOU, F.A. TOBAGI, AND M.J. KARAM. Assessment of VoIP quality over Internet backbones. In *INFOCOM 2002. Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, **1**, pages 150–159 vol.1, 2002. 112

[92] PEDRO MERINO AND ALBERTO SALMERÓN. Analysis and optimization of video download on mobile devices. In *XX Jornadas de Telecom I+D. Valladolid 27, 28 y 29 septiembre 2010 (CD-ROM)*, December 2010. 4, 186

[93] PEDRO MERINO AND ALBERTO SALMERÓN. Combining SPIN with ns-2 for protocol optimization. In *17th International SPIN Workshop on Model Checking of Software, SPIN 2010*, **6349**, pages 40–57. Springer, 2010. 4, 98, 186

[94] J. MÜLLER, M. LORENZ, F. GELLER, A. ZEIER, AND H. PLATTNER. Assessment of communication protocols in the EPC Network - replacing textual SOAP and XML with binary google protocol buffers encoding. In *Industrial Engineering and Engineering Management (IE EM), 2010 IEEE 17Th International Conference on*, pages 404–409, 2010. 57

[95] MADANLAL MUSUVATHI, DAVID Y. W. PARK, ANDY CHOU, DAWSON R. ENGLER, AND DAVID L. DILL. CMC: A pragmatic approach to model checking real code. In *In Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, 2002. 25

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

[96] PAUL MUTTON. Jibble Web Server. `http://www.jibble.org/jibblewebserver.php`, February 2014. 141

[97] BRIAN D. NOBLE, M. SATYANARAYANAN, GIAO T. NGUYEN, AND RANDY H. KATZ. Trace-based mobile network emulation. *SIGCOMM Comput. Commun. Rev.*, **27**:51–61, October 1997. 129

[98] GEORGES NOGUEIRA, BRUNO BAYNAT, AND AHMED ZIRAM. An efficient analytical model for QoS engineering in mobile cellular networks. In *WOWMOM '08: Proceedings of the 2008 International Symposium on a World of Wireless, Mobile and Multimedia Networks*, pages 1–12, Washington, DC, USA, 2008. IEEE Computer Society. 122

[99] G. NORMAN AND V. SHMATIKOV. Analysis of probabilistic contract signing. *Journal of Computer Security*, **14**[6]:561–589, 2006. 10

[100] DORON PELED. Combining partial order reductions with on-the-fly model-checking. In DAVIDL. DILL, editor, *Computer Aided Verification*, **818** of *Lecture Notes in Computer Science*, pages 377–390. Springer Berlin Heidelberg, 1994. 67

[101] GIUSEPPE PIRO, NICOLA BALDO, AND MARCO MIOZZO. An lte module for the ns-3 network simulator. In *Proceedings of the 4th International ICST Conference on Simulation Tools and Techniques*, pages 415–422. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2011. 149

[102] AMIR PNUELI. The temporal logic of programs. pages 46–57, oct. 1977. 12, 13, 19, 178

[103] STEVEN PROCTER. Java NFS Server. `http://www.void.org/~steven/jnfs/`, February 2014. 140

[104] J.P. QUEILLE AND J. SIFAKIS. Specification and verification of concurrent systems in CESAR. In MARIANGIOLA DEZANI-CIANCAGLINI AND UGO MONTANARI, editors, *International Symposium on Programming*, **137** of *Lecture Notes in Computer Science*, pages 337–351. Springer Berlin Heidelberg, 1982. 9

[105] ALBERTO SALMERÓN AND PEDRO MERINO. On-the-fly VoIP call quality evaluation with improved E-model. In *Proceedings of the 8th Performance Monitoring, Measurement and Evaluation of Heterogeneous Wireless and Wired Networks Workshop, PM2HW2N 2013*, pages 145–151. ACM, November 2013. 4, 109, 118, 186

[106] DOUGLAS C. SCHMIDT. Pattern Languages of Program Design. chapter Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

Handler Dispatching, pages 529–545. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1995. 94

[107] DOUGLAS C. SCHMIDT AND STEPHEN D. HUSTON. *C++ Network Programming Vol. 1: Mastering Complexity with ACE and Patterns*. Addison-Wesley, 2nd edition, 2002. 94

[108] DOUGLAS C. SCHMIDT AND STEPHEN D. HUSTON. *C++ Network Programming Vol. 2: Systematic Reuse with ACE and Frameworks*. Addison-Wesley, 2002. 94

[109] BRUCE SCHNEIER. *Applied cryptography (2nd ed.): protocols, algorithms, and source code in C*. John Wiley & Sons, Inc., New York, NY, USA, 1995. 71

[110] XIAOCHUAN SHEN, ADRIAN WONFOR, RICHARD PENTY, AND IAN WHITE. Receiver Playout Buffer Requirement for TCP Video Streaming in the presence of Burst Packet Drops. In *London Communications Symposium 2009*, 2009. 103

[111] A. SOBEIH, M. D'AMORIM, M. VISWANATHAN, D. MARINOV, AND J. C. HOU. Assertion Checking in J-Sim Simulation Models of Network Protocols. *SIMULATION*, **86**[11]:651–673, October 2009. 25

[112] A. SOBEIH, J.C. HOU, LU-CHUAN KUNG, N. LI, HONGHAI ZHANG, WEI-PENG CHEN, HUNG-YING TYAN, AND HYUK LIM. J-Sim: a simulation and emulation environment for wireless sensor networks. *Wireless Communications, IEEE*, **13**[4]:104–119, 2006. 25

[113] PETER SOROTOKIN. FTP Server in Java. `http://peter.sorotokin.com/ftpd/ftpd.html`, February 2014. 137

[114] YE TIAN, KAI XU, AND N. ANSARI. TCP in wireless environments: problems and solutions. *Communications Magazine, IEEE*, **43**[3]:S27–S32, March 2005. 98

[115] MOSHE Y. VARDI AND PIERRE WOLPER. Automata-theoretic techniques for modal logics of programs. *Journal of Computer and System Sciences*, **32**[2]:183–221, 1986. 22, 137

[116] MOSHE Y. VARDI AND PIERRE WOLPER. Reasoning about infinite computations. *Information and Computation*, **115**:1–37, 1994. 13, 15

[117] WILLEM VISSER, KLAUS HAVELUND, GUILLAUME BRAT, SEUNGJOON PARK, AND FLAVIO LERDA. Model Checking Programs. *Automated Software Engineering*, **10**[2]:203–232, 2003. 25, 28, 144

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA

[118] PIERRE WOLPER AND DENIS LEROY. Reliable hashing without collision detection. In COSTAS COURCOUBETIS, editor, *Computer Aided Verification*, **697** of *Lecture Notes in Computer Science*, pages 59–70. Springer Berlin Heidelberg, 1993. 71

[119] TAO YE, HEMA T. KAUR, SHIVKUMAR KALYANARAMAN, AND MURAT YUKSEL. Large-scale network parameter configuration using an on-line simulation framework. *IEEE/ACM Trans. Netw.*, **16**[4]:777–790, 2008. 26, 27

[120] FUYUAN ZHANG, ZHENGWEI QI, HAIBING GUAN, XUEZHENG LIU, MAO YANG, AND ZHENG ZHANG. FiLM: A Runtime Monitoring Tool for Distributed Systems. In *Secure Software Integration and Reliability Improvement, 2009. SSIRI 2009. Third IEEE International Conference on*, pages 40–46, july 2009. 30

[121] LIREN ZHANG, LI ZHENG, AND KOH SOO NGEE. Effect of delay and delay jitter on voice/video over IP. *Computer Communications*, **25**[9]:863–873, 2002. 129

Publicaciones y Divulgación Científica
UNIVERSIDAD DE MÁLAGA