THÈSE

Pour obtenir le grade de

DOCTEUR DE LA
COMMUNAUTÉ UNIVERSITÉ GRENOBLE ALPES

Spécialité : Informatique

Arrêté ministériel : 25 mai 2016

Présentée par

Gianluca BARBON

Thèse dirigée par **Gwen SALAUN**, UGA
et codirigée par **Vincent LEROY**, Université Grenoble Alpes / LIG

préparée au sein du **Laboratoire Laboratoire d'Informatique de Grenoble**
dans **l'École Doctorale Mathématiques, Sciences et technologies de l'information, Informatique**

Débogage de modèles comportementaux par analyse de contre-exemple

Debugging of Behavioural Models using Counterexample Analysis

Thèse soutenue publiquement le **14 décembre 2018**,
devant le jury composé de :

Monsieur GWEN SALAÜN
PROFESSEUR, UNIVERSITE GRENOBLE ALPES, Directeur de thèse
Madame OLGA KOUCHNARENKO
PROFESSEUR, UNIVERSITE DE FRANCHE-COMTE, Rapporteur
Monsieur FRANCISCO DURAN
PROFESSEUR, UNIVERSITE DE MALAGA - ESPAGNE, Rapporteur
Monsieur STEFAN LEUE
PROFESSEUR, UNIVERSITE DE CONSTANCE - ALLEMAGNE, Examinateur
Monsieur ROLAND GROZ
PROFESSEUR, GRENOBLE INP, Président
Monsieur VINCENT LEROY
INGENIEUR, GOOGLE A ZURICH - SUISSE, Co-directeur de thèse

Acknowledgements

First of all, I would like to express my special appreciation and thanks to my thesis director, Gwen Salaün, for his availability and guidance throughout my Ph.D., and for his valuable comments in all my writings and rehearsals. I am also deeply grateful to my thesis co-supervisor Vincent Leroy, in particular for his precious advices and help in the tool development. Second, I would like to express my gratitude to my reviewers, Olga Kouchnarenko and Francisco Durán, and to my examiners, Stefan Leue and Roland Groz, for their precious remarks on my manuscript and for accepting to be part of the committee.

I am also grateful to all the CONVECS team members for having welcomed me as part of their family. In particular, I would like to thank Radu Mateescu, Frederic Lang, Wendelin Serwe and Hubert Garavel for both technical and theoretical advices and for the all the insightful discussions we had. I would also like to thank Myriam Etienne, our team assistant, for her kindness and the support with all the administrative concerns. Many thanks are due to all the other past CONVECS team members I had the opportunity to meet, who have helped me in one way or another.

A special thank goes to my office mates Lina Marsso and Ajay Krishna, for helping me and for making the atmosphere enjoyable even in stressful moments. I also would like to thank Emmanuel Yah Lopez for contributing in the development of a part of our tool.

Sincere thanks go out to all my friends, in Italy and in France, who have accompanied me during this adventure. Someone in the laboratory, someone else in everyday life, you all gave me a help. Particular thanks are due to all the wonderful people I met in Grenoble, who shared with me a lot of funny (and less funny) moments during these three years.

Finally, last but not least, I would like to express my sincere gratitude to my family for supporting me every day, in the good and in the bad times. It is also thanks to you all that this thesis has finally come to light.

Abstract

Model checking is an established technique for automatically verifying that a model satisfies a given temporal property. When the model violates the property, the model checker returns a counterexample, which is a sequence of actions leading to a state where the property is not satisfied. Understanding this counterexample for debugging the specification is a complicated task for several reasons: (i) the counterexample can contain a large number of actions; (ii) the debugging task is mostly achieved manually; (iii) the counterexample does not explicitly point out the source of the bug that is hidden in the model; (iv) the most relevant actions are not highlighted in the counterexample; (v) the counterexample does not give a global view of the problem.

This work presents a new approach that improves the usability of model checking by simplifying the comprehension of counterexamples. Our solution aims at keeping only actions in counterexamples that are relevant for debugging purposes. This is achieved by detecting in the models some specific choices between transitions leading to a correct behaviour or falling into an erroneous part of the model. These choices, which we call "neighbourhoods", turn out to be of major importance for the understanding of the bug behind the counterexample. To extract such choices we propose two different methods. One method aims at supporting the debugging of counterexamples for safety properties violations. To do so, it builds a new model from the original one containing all the counterexamples, and then compares the two models to identify neighbourhoods. The other method supports the debugging of counterexamples for liveness properties violations. Given a liveness property, it extends the model with prefix / suffix information w.r.t. that property. This enriched model is then analysed to identify neighbourhoods.

A model annotated with neighbourhoods can be exploited in two ways. First, the erroneous part of the model can be visualized with a specific focus on neighbourhoods, in order to have a global view of the bug behaviour. Second, a set of abstraction techniques we developed can be used to extract relevant actions from counterexamples, which makes easier their comprehension. Our approach is fully automated by a tool we implemented and that has been validated on real-world case studies from various application areas.

Résumé

Le *model checking* est une technique établie pour vérifier automatiquement qu'un modèle vérifie une propriété temporelle donnée. Lorsque le modèle viole la propriété, le *model checker* retourne un contre-exemple, i.e., une séquence d'actions menant à un état où la propriété n'est pas satisfaite. Comprendre ce contre-exemple pour le débogage de la spécification est une tâche compliquée pour plusieurs raisons: (i) le contre-exemple peut contenir un grand nombre d'actions; (ii) la tâche de débogage est principalement réalisée manuellement; (iii) le contre-exemple n'indique pas explicitement la source du bogue qui est caché dans le modèle; (iv) les actions les plus pertinentes ne sont pas mises en évidence dans le contre-exemple; (v) le contre-exemple ne donne pas une vue globale du problème.

Ce travail présente une nouvelle approche qui rend plus accessible le *model checking* en simplifiant la compréhension des contre-exemples. Notre solution vise à ne garder que des actions dans des contre-exemples pertinents à des fins de débogage. Pour y parvenir, on détecte dans les modèles des choix spécifiques entre les transitions conduisant à un comportement correct ou à une partie du modèle erroné. Ces choix, que nous appelons *neighbourhoods*, se révèlent être de grande importance pour la compréhension du bogue à travers le contre-exemple. Pour extraire de tels choix, nous proposons deux méthodes différentes. La première méthode concerne le débogage des contre-exemples pour la violations de propriétés de sûreté. Pour ce faire, elle construit un nouveau modèle de l'original contenant tous les contre-exemples, puis compare les deux modèles pour identifier les *neighbourhoods*. La deuxième méthode concerne le débogage des contre-exemples pour la violations de propriétés de vivacité. À partir d'une propriété de vivacité, elle étend le modèle avec des informations de préfixe / suffixe correspondants à cette propriété. Ce modèle enrichi est ensuite analysé pour identifier les *neighbourhoods*.

Un modèle annoté avec les *neighbourhoods* peut être exploité de deux manières. Tout d'abord, la partie erronée du modèle peut être visualisée en se focalisant sur les *neighbourhoods*, afin d'avoir une vue globale du comportement du bogue. Deuxièmement, un ensemble de techniques d'abstraction que nous avons développées peut être utilisé pour extraire les actions plus pertinentes à partir de contre-exemples, ce qui facilite leur compréhension. Notre approche est entièrement automatisée par un outil que nous avons implémenté et qui a été validé sur des études de cas réels dans différents domaines d'application.

Contents

Chapter 1

Introduction

Recent computing trends are promoting the development of concurrent and distributed applications, which are used in various domains. Here are listed some examples. Cyber-physical systems involve many components such as controllers, sensors, or processors, which are supposed to interact together to fulfil some specific functions. Software and middleware technologies connect smart grids and smart meters in order to anticipate and optimize energy consumption. Service Oriented Computing enables the implementation of value-added Web accessible software systems that are composed of distributed services. Cloud computing leverages hosting platforms based on virtualization, and promises to deliver computational resources on demand via a computer network. Internet of Things networking model allows physical objects to be seamlessly integrated into the information network, and to become active participants in business processes.

The intrinsically parallel and distributed nature of these application makes them complex to design and develop, and it favours the introduction of subtle coding errors, called bugs. Bugs are defects of software which prevent the correct behaviour of the system. According to [Kid98], the term bug was already used to name a *"fault in the working of any electrical apparatus"* in the end of the 19th century, and was also used with the same meaning by the well-known American inventor Thomas Edison. This term was later adopted by the newborn computer engineering community in the late 1940s. After some decades, we are still far from proposing techniques and tools avoiding the existence of bugs in a system under development. However, it is nowadays possible to automatically chase and find bugs that would have been very difficult, or even impossible, to detect manually. The process of finding and resolving bugs is commonly called *debugging*. The verb *to debug* spread out among professionals in the computer science field between the late 1940s and early 1950s, to indicate the removal of *"malfunctioning conditions from a computer or an error from a routine"* [Kid98]. Grace Hopper, while working on the IBM Mark I and II computers at Harvard, was among the first using the term *debug* in such sense. The debugging step is today one of the most important steps in the software development.

1.1 Context

The debugging process is still a challenging task for a developer, since it is difficult for a human being to understand the behaviour of all the possible executions of concurrent systems. Moreover, bugs can be hidden inside parallel behaviours. Because of this, the debugging task still takes a considerable part of the whole time spent in software development. Thus, there is a need for automatic techniques that can help the developer in detecting and understanding those bugs, reducing the cost of software development.

Various methods have been developed in order to debug software, e.g., control flow analysis, testing, slicing, static analysis. Model checking [CGP01, BK08] is one of these techniques, with a specific focus on the verification of concurrent systems. This approach takes as input a model and a property. The model, e.g., a Labelled Transition System (LTS), describes all the possible behaviours of a concurrent program and is obtained by compilation from a specification of the system (expressed with higher-level textual specification languages such as process algebra). The property represents the requirements of the system and is usually expressed with a temporal logic. Given a model and a property, a model checker verifies whether the model satisfies the property. When the model violates the property, the model checker returns a counterexample, which is a sequence of actions leading to a state where the property is not satisfied.

1.2 Motivations

While a lot of work has been done on detecting the existence of program faults, much less has been done on localizing such faults [GSB07]. Although model checking techniques automatically find bugs in concurrent systems, it is still difficult to interpret and understand the returned counterexamples for several reasons:

(i) The counterexample can be *lengthy*, since it may contain hundreds (even thousands) of actions [BBC$^+$12, LB13]. Large real-world specifications are often long and complex, implying also long and complex counterexamples [BBC$^+$12].

(ii) The debugging task is mostly achieved *manually*, since satisfactory automatic debugging techniques do not yet exist. It is usually up to the developer to understand the counterexample to locate the error [CGS04, GCKS06]. While a counterexample should ideally be succinct and comprehensible to be readable by the developer [GK05], its length and complexity force the developer to spend considerable time in examining it [LB13, GCKS06]. The counterexample analysis is thus a challenging and tedious task, which requires a significant effort to the developer [GV03, GSB10, BBC$^+$12]. Note that the non-automated nature of the debugging not only takes a lot of time, but it may also lead to errors in the bug comprehension [LB13]. Moreover, understanding the counterexample in concurrent systems might be even more difficult because of interleavings. The interleaving semantics implies non-deterministic choices between the processes, while a developer is normally used to think sequentially [LB13].

(iii) The counterexample does not provide sufficient information to understand the bug [GV03]. As a matter of fact, a counterexample is only a symptom of a bug which shows its existence, but it does not explain its *sources* [WYIG06]. The possible causes of the bug are hidden in the model, and may be represented by choices whose existence is not evident in the sole counterexample. Searching for the actual causes of a bug remains one of the most time consuming tasks in debugging [Zel09].

(iv) The actions in the counterexample seem to have the same importance even if it is not the case [JRS02]. For instance, even if the last action of a counterexample is located where the property become unsatisfied, the cause of the property violation might be linked to actions which are located earlier in the counterexample. The importance of these relevant actions is *not highlighted* in the counterexample.

(v) A counterexample is only one of the traces that lead to the bug [JRS02], thus a single counterexample might not reveal all the circumstances in which the failure arises. Even if the developer is able to fix the error expressed by the counterexample, the model checker might produce other counterexamples in successive runs [BNR03]. A single counterexample thus *does not give a global view* of the problem.

1.3 Approach

This work aims at developing a new approach for simplifying the comprehension of counterexamples and thus favouring usability of model checking techniques. More precisely our approach aims to automatically improve the comprehension of counterexamples by taking only into account relevant actions and providing more exhaustive information about the source of the bug, while keeping a global view of the faulty model. To do so, we highlight some of the actions in the counterexample that are of prime importance. These actions make the specification go from a (potentially) correct behaviour to an incorrect one. Thus, they correspond to decisions or choices in the model that are of particular interest because they can provide an explanation of the source of the bug. Once these choices have been identified, they can be used for building simplified versions of the counterexample, for instance keeping only actions that are relevant from a debugging perspective.

More precisely, in order to detect these choices, we need to categorize the transitions in the model according to the behaviour of the system they lead to. We can thus recognize three types of transitions; (a) *correct transitions* are transitions that only lead to correct behaviours in the model; (b) *incorrect transitions* are transitions that only lead to incorrect behaviours in the model; (c) *neutral transitions* are transitions that lead to both correct and incorrect behaviours. We call *neighbourhood* a choice in the model between two (or more) different transitions types. A neighbourhood consists of the set of incoming and outgoing transitions of the states where these decision points are located. Those sets of transitions identify specific parts of the specification that may explain the appearance of the bug and are therefore meaningful from a debugging perspective. The nature of outgoing transitions (correct, incorrect and neutral) allows us to classify neighbourhoods in different types.

To extract transitions, and consequently neighbourhoods, from a model, we propose two different approaches. The first one is called *Counterexample LTS* approach, and it has been specifically designed to handle safety properties. Safety properties state that "*something bad never happens*". This method produces all the counterexamples from a given model and compare them with the correct behaviours of the model to later identify neighbourhoods. The second approach is called *Prefix/Suffix* approach, and it has been built to handle liveness properties, which are properties used to state that "*something good eventually happens*". It focuses on the analysis of property's actions execution in each state of the model to locate neighbourhoods.

More precisely, the Counterexample LTS approach first extracts all the counterexamples from the original model containing all the executions. This procedure is able to collect all the counterexamples in a new LTS, maintaining a correspondence with the original model. To do this, an LTS of the formula that represents the property is first created. Then, the synchronous product between the original LTS and the LTS of the formula is performed. The resulting LTS is called *counterexample LTS*. Its states are simulated by the ones in the original LTS and it only contains traces that violates the property. Second, an algorithm is defined to identify actions in the frontier where counterexamples and correct behaviours, that share a common prefix, split in different paths. The states of the two LTSs that belong to this frontier are compared to extract the differences in terms of outgoing transitions between these states. The original LTS can contain outgoing transitions from a state that do not appear in the corresponding state of the counterexample LTS. This means that they belong to correct paths of the original LTS, which represent behaviours that do not violate the given property, and thus represent correct transitions. By searching for paths in the counterexample LTS that do not contain any correct transitions we are also able to identify incorrect transitions (and consequently, neutral ones). Finally, when all the transitions have been typed, we can analyse the incoming and outgoing transitions of each state in the LTS to determine whether it is a neighbourhood or not.

The Prefix/Suffix approach allows to handle liveness properties. More precisely, we have a specific focus on inevitability properties, which are one of the classes of liveness properties most used by developers in practice [DAC99]. This second approach takes as input an LTS and a liveness property in the form of a sequence of inevitable actions. In a first step it enhances each state of the LTS with prefix/suffix information about the actions belonging to the property that have already been or remain to be executed. Prefix and suffix information is successively used to infer the type (correct, incorrect or neutral) of each transition. Given a transition, the prefix information on its source state and the suffix information on its destination state are combined and compared to the sequence of inevitable actions. If such combination leads to always respecting the sequence of inevitable actions, the transition is labelled as correct. If the sequence is never respected, the transition is labelled as incorrect. If the combination of prefix and suffix information reveals that the sequence is respected only in some cases, the transition is labelled as neutral. When all transitions have been typed, incoming and outgoing transitions of each state in the LTS can be analysed to detect neighbourhoods.

We can finally exploit the LTS where neighbourhoods have been discovered to extract precise information related to the bug, through the use of visualization and abstraction techniques. We propose a debugging procedure to exploit neighbourhood which consists in first making use of a global visual feedback of the model containing neighbourhoods, and then allowing the developer to go into details and focus on single counterexamples.

More precisely, the developer can first exploit 3D visualization techniques we have developed to obtain a global view of the model, where correct/incorrect/neutral transitions and neighbourhoods are highlighted using different colours. We implemented this visualization method in a tool which provides several functionalities to facilitate the manipulation of those models (forward/backward step-by-step animation, counterexample visualization, zoom in/out, etc.).

In a subsequent step, given a counterexample, the developer can exploit several abstraction techniques defined on top of the notion of neighbourhood. These techniques aim at removing irrelevant parts of the counterexample and highlighting relevant ones to simplify its comprehension. Abstraction techniques can be divided into techniques that are independent of the model and dependent on the model. The former set of techniques are the ones that do not require an additional input from the user to give a result. Conversely, the latter set consists of techniques which exploit a user input (e.g., a pattern of actions) to perform the analysis. An example of the first set is the counterexample abstraction, which consists of simplifying a given counterexample by keeping only actions which belongs to neighbourhoods, thus making the debugging process easier by reducing the size of the counterexample. An example of the second set of techniques is the search for the shortest path from the initial node to a neighbourhood matching a pattern of actions provided by the user. This abstraction technique is useful to focus on specific actions of the model and to check whether they are relevant (or not) from a debugging perspective.

1.4 Contributions

This thesis presents an approach to extract relevant choices in a model that can provide an explanation for the bug. As a summary, the main contributions of this thesis are the following ones:

- The definition of the notion of choice inside the LTS, called neighbourhood, between correct and incorrect behaviours. Several notions of neighbourhoods have been defined depending on the type of transitions located at such a state (correct, incorrect or neutral transitions).

- The development of a method to generate, given a model and a safety property, the counterexample LTS, that is a model containing all the counterexamples.

- The development of algorithms that, given a model and a liveness property in the form of a sequence of inevitable actions, are able to evaluate if the traces that go through each state of the model represent prefixes and/or suffixes of the property.

- The implementation of our approach in a tool, called CLEAR, which has been published online [cle]. This tool consists of three main modules: a computation module, which is responsible for the detection of transition types and for the neighbourhood computation; a 3D visualization module, which allows the developer to graphically observe the whole model and see how neighbourhoods are distributed over that model; an analysis module, which implements the abstraction techniques.

- The validation of our approach on real-world case studies from various application areas. Our experiments show that our approach, by exploiting the notion of neighbourhood together with the set of provided visualization and abstraction techniques, simplifies the comprehension of the bug. Moreover, experiments carried on with the 3D visualization technique allowed us to identify some interesting visualization patterns that correspond to typical cases of bugs. As a result, the 3D visualization technique can be used for visual debugging in order to identify the bug by looking at the graphical representation of the model extended with neighbourhoods.

These contributions have been (partially) published in two conference papers. The first one [BLS17] describes a preliminary version of our approach for counterexample analysis of safety property violations. The second one [BLS18] presents the approach working with liveness property violations, with an improved version of neighbourhoods.

1.5 Thesis Structure

The rest of this thesis is organized as follows:

Chapter 2 overviews related work, presenting various types of debugging techniques, with a particular focus on trace-based explanation approaches and testing-based fault localization. We also describe bug visualization techniques and other alternative approaches in two dedicated sections. At the end of the chapter we position our approach with respect to the discussed related work.

Chapter 3 illustrates the scientific background needed to understand notions presented in the rest of this work. In particular, it first introduces behavioural model (LTS model) and model checking notions. The LNT language, which is the process algebra we adopt to specify a system, is also described. At the end of the chapter we present some definitions and algorithms that we will use on LTSs.

Chapter 4 surveys all the steps of our debugging approach. It first details the transitions types, the neighbourhood notion and gives a neighbourhood taxonomy. Second, a methodology for debugging a faulty LTS is presented, exploiting visualization and abstraction techniques on top of neighbourhoods.

Chapter 5 presents the Counterexample LTS approach for handling safety properties violations. It first describes the steps for generating the counterexample LTS. Then it details how states of the original LTS and of the counterexample LTS match, in order to later detect neighbourhoods. Finally, the step for typing transitions is presented.

Chapter 6 presents the Prefix / Suffix approach that we built for handling liveness properties violations. It first details the algorithms to compute prefixes and suffixes in a model following the given sequence of inevitable actions. Second, it illustrates the method that recognizes transitions types by exploiting the prefix and suffix information.

Chapter 7 describes the implementation of our approach in the CLEAR tool [cle], presenting in detail the three tool modules that performs neighbourhood detection, 3D visualization and abstraction techniques, respectively.

Chapter 8 presents the application of our approach on real-word examples. First, it describes the visualization techniques for erroneous behavioural models by illustrating them on some examples. Second, it details the use of abstraction techniques on four case studies. An empirical study is also illustrated at the end of the chapter to validate our approach with a set of developers.

Chapter 9 concludes this work, summarising our main contributions, and discusses future perspectives.

Chapter 2

Related Work

In this chapter, we survey research works providing techniques for supporting the debugging of specifications and programs, with a particular focus on trace-based explanation approaches and testing-based fault localization.

Fault localization for program debugging has been an active topic of research for many years in the software engineering community. Several options have been investigated such as static analysis, slice-based methods, statistical methods, or machine learning approaches. In 1997 Lieberman, in his introduction to the special issue on The Debugging Scandal [Lie97] highlighted the potential of debugging, warning about the lack of consideration (for the time) by the computer science community. In his introduction he encouraged the computer science community to spend more time on debugging, and to use the new achievements in software and hardware technology in this task. For instance, he proposed to use the new improvements on computer graphics to visualize the behaviour of the programs, in order to help developers in relating the static nature of the code to the dynamic behaviour of a program. Citing articles appearing in that special issue, he suggested the adoption of innovative ideas in the debugging process, like the use of software visualization and of collaborative debugging. Lieberman finally concluded by trying to convince the industry to adopt debugging techniques, highlighting the lack of consideration of debugging tools.

In the successive years debugging tools have been widely adopted by the community. An exhaustive survey about existing techniques can be found in [WGL^{+}16]. In addition to the techniques we will discuss in the rest of this chapter, [WGL^{+}16] also details fault localization with multiple bugs, coincidental correctness (when a program has produced one or more bugs during execution but, for some coincidence, the output of the program remains correct) and combination of multiple fault localization techniques. A relevant reference which gives a general perspective about debugging is the book of Zeller "Why Programs Fail - A Guide to Systematic Debugging" [Zel09], which proposes to show that automatic and manual debugging techniques can be used as systematic methods to find and fix bugs in computer program.

Note that some of the works we present in this chapter rely on causality analysis, which aims at relating causes and effects to help in debugging faults in (possibly concurrent) systems. This kind of analysis relies on a notion of counterfactual reasoning, where alternative executions of the system are derived by assuming changes in the program. Counterfactual reasoning has been first introduced by Lewis [Lew73], and then it has been improved by Halpern and Pearl with the introduction of the so-called actual cause conditions [HP05]. Some relevant examples of research works that rely on causality analysis are the line of work of Goössler et al. [GMR10, GM13, GM15, WAK$^+$13, WGG$^+$15, GA14, GS15] and the one of Leitner-Fischer and Leue [LL13, LL14, BHK$^+$15]. It is worth noting that our approach does not have a focus on causality analysis, instead we focus on the analysis of choices made in the model and their impact on the validity of the given property.

Before presenting in detail the major techniques in debugging of concurrent systems, we make two brief discussions about a proposed taxonomy of faults and about faults categorization works.

A Taxonomy of Faults In [ALRL04] Avizienis et al. propose a taxonomy, making a precise distinction between the terms *failure*, *error* and *faults*. A failure is described as an event in which a service deviates from its correct behaviour. An error is a condition of a system that may lead to a subsequent system failure. A fault (or bug) is the underlying cause of an error, and can be divided into internal or external of a system. In [ALRL04] the authors also discuss about some open questions in bug detection. In particular, they state that the detection of bugs could benefit from a better understanding of real world concurrency bug characteristics. However, the authors are aware that such characteristic are still far to be catalogued, since real world bugs are usually under-reported by developers, and are often not easy to understand. In our work we do not explicitly make a distinction between failures, error and faults, as Avizienis did in [ALRL04]. Note that we will use the words fault and bug as synonyms in the rest of this work.

Faults Categorization (generic) Recently, various research works have made an effort to categorize faults. A general software fault classification is described in [GT05], where the authors distinguish bugs between *bohrbugs* and *mandlebugs*. Bohrbugs are detailed as bugs that can be easily isolated and that manifest under precise conditions, while mandlebugs are bugs which are difficult to isolate and whose propagation and/or activation are complex. Mandlebugs, in turn, comprehend *heisenbugs* and *ageing-related bugs*. Heisenbugs are defined as a sort of elusive faults, which “stops causing a failure” or which can manifest in different ways if the developer tries to isolate them. Ageing-related bugs are defined as faults that cause an accumulation of errors in the system. Note that two classes, heisenbugs and ageing-related bugs, may overlap.

Faults Categorization (in concurrent systems) In [FNU03] Farchi et al. present a preliminary work to define bug characteristics in concurrent systems. Note that this study has been built by observing programs that were intentionally buggy, not from real bugged software. A comprehensive study of concurrency bugs characteristics is also presented in [LPSZ08], where the authors examine bugs in four large open source applications in order to propose some categorizations of concurrency bugs. They first recognize two prevalent categories of non-deadlock concurrency bugs: *data-races*, which are conflicting accesses to one shared variable, and *deadlocks*, which occur when there exists a circular wait between different processes. Second, they search for and try to categorize patterns in bug, identifying two main patterns in non-deadlock concurrency bugs: *atomicity violations*, that take place when concurrent executions violate the atomicity of a given code portion, and *ordering violations*, which take place when the expected order of two memory accesses is flipped. However, according to the authors their work does not intend to give general conclusions about concurrency bugs, since it only focused on four applications and their programming languages.

In our work we do not propose a categorization of concurrent faults. However, in Chapter 8 we present some experiments in which the visual rendering of bugs can lead to a sort of bug categorization, exploiting the specific structure that characterize the bug in the rendering. Note that our categorization does not propose to be exhaustive, but only suggests that some bugs might be easily recognized by looking at their visual representation.

We will now present related work, divided into two main sets: trace explanation and fault localization using testing. We will also describe bug visualization techniques and alternative approaches (which do not rely neither on trace-based nor on test-based techniques) in two dedicated sections.

2.1 Trace Explanation

An important line of research focuses on interpreting traces (execution traces or counterexamples generated from model checking) and favouring their comprehension. In particular, techniques based on counterexample analysis represent the line of research that is closer to our work. This section is divided in various paragraphs, each one dealing with a specific trace-based approach. The first three paragraphs describe techniques which focuses on the interpretation and on the explanation of the counterexample (see for instance [JRS02, GV03]). The central part of the section presents techniques based on pattern mining, distance metrics and minimal changes in a program. The last part of the chapter describe methods that rely on causality notions applied to trace analysis in order to give an explanation of the fault.

Game-theoretic Counterexample Interpretation In [JRS02] the authors propose a game-theoretic method to interpret counterexamples traces from liveness properties by partitioning them into fated and free segments. In particular, the computation of fated segments is formulated in the form of two player reachability games between the system and the environment. Fated segments represent inevitability w.r.t. the failure, pointing out progress towards the bug, while free segments highlight the possibility to avoid the bug. The proposed approach classifies states in a state-based model in different layers (which represent distances from the bug) and produces a counterexample annotated with segments by exploring the model. Both our work and [JRS02] aim at building an explanation from the counterexample. However, our method focuses on locating branching behaviours that affect the property satisfaction whereas their approach produces an enhanced counterexample where inevitable events (w.r.t. the bug) are highlighted.

Multiple Variations of a Counterexample In [GV03] Groce and Visser propose automated methods for the analysis of multiple variations of a counterexample, in order to identify portions of the source code crucial to distinguishing failing and successful runs. These variations can be distinguished between executions that produce an error (negatives) and executions that do not produce it (positives). Note that the set of negatives is defined in such a way to focus on errors that are linked by similar behaviour. The authors then propose various analysis to extract common features and differences between the two sets in order to provide feedbacks to the user. Three different extraction methods are proposed: a transition analysis, an invariant analysis and a transformation analysis. The third one in particular appears more interesting since, by applying transformation of positives into negatives, it allows understanding concurrency problems, i.e. unexpected interleavings. Their approach has been implemented in the Java PathFinder [VHBP00] model checker and takes as input compiled Java programs. Ball et al. present in [BNR03] a method similar to the one proposed in [GV03], implemented in a tool called SLAM [BR01]. Their approach takes as input C programs in the form of control flow graphs, calls a model checker as a subroutine to localise the error cause and generates a counterexample for each cause. It then extracts the transitions of the counterexample that do not appear in any correct trace, thus representing program statements that are probably related to the cause of the bug. Similarly to our work, the work presented by Groce and Visser in [GV03] also wants to better explain the counterexample. However, while our approach is generic w.r.t the language that produces the LTS, they have a specific focus on Java bytecode for their analysis. Moreover our approach has a global view on the whole LTS, while they only focus on a single counterexample and its variations. As for the work of Ball et al. in [BNR03], we do not need to perform multiple calls to the model checker, as we reason on the whole model. Moreover, while in [BNR03] the authors have a specific focus on safety properties, we can also handle liveness ones.

Concise Counterexamples In [RS04] the authors aim to ease counterexample understanding in SAT-based Bounded Model Checking. Their approach allows extracting a succinct counterexample, which allows identifying interesting events that contribute to the cause of the failure. In particular, it finds minimal satisfying assignments to values of variables of the input formulas in order to isolate the events responsible for the property violation. The work of Ermis et al. [ESW12] also aims at giving a concise explanation of a counterexample. This work takes as input a counterexample and formulas describing the initial and the expected output states for the counterexample. The authors then use the notion of error invariants, which are formulas that over-approximates the reachable states at a given point of an error trace. By using these error invariants, their approach allows identifying irrelevant transitions in an error trace and thus can provide a compact representation of that error trace. This approach has been implemented on top of the SMTInterpol tool [CH10]. Methods which produce a concise counterexample, like [ESW12, RS04], are partly similar to our approach, since their goal is to ease counterexample understanding. However note that those works have a different point of view to counterexample analysis with respect to us, since while we focus on extracting relevant actions on counterexamples, they aim at slicing the counterexample by computing an invariant formula [ESW12] or by computing minimal assignments to variables of the input formula [RS04].

Pattern Mining in Trace Analysis Some recent works perform counterexample analysis by applying pattern mining techniques, explicitly taking into account concurrency. In [BWW14], sequential pattern mining is applied to execution traces for revealing unforeseen interleavings that may be a source of error, through the adoption of the well-known mining algorithm CloSpan [YHA03]. The authors aim at identifying event sequences that frequently occurs in faulty execution traces. This work deals with various typical issues in the analysis of concurrent models, for instance the problem of increasing length of traces and the introduction of spurious patterns when abstraction methods are used. CloSpan is also adopted in [LB13], where Leue and Befrouei apply sequential pattern mining to traces of counterexamples generated from a model using the SPIN model checker. By doing so, they are able to reveal unforeseen interleavings that may be a source of error. The approach presented in [LB13] is able to analyse concurrent systems and to extract sequences of events for identifying bugs, thus representing one of the closest results to our work. This approach is an enhanced version of the one proposed in [LB12] by the same authors, where only contiguous events could be detected. Some works make use of pattern mining approaches to reason on traces, as achieved in [BWW14, LB12, LB13]. In some cases the use of pattern mining approaches to reason on traces may result more scalable for large systems, and they can be helpful in cases in which it is not possible to obtain a model of the system. In [LB12] the authors also make a precise comparison with the counterexample analysis approach of Groce and Visser detailed in [GV03], which is the most closely related work to theirs, showing that their approach requires to the user less effort to understand the cause of a bug in a given case study. It is worth noting that reasoning on traces can also induce several issues. For instance, the handling of looping behaviours is non-trivial and may result in the generation of infinite traces or of an infinite number of traces. Coverage is another problem, since a high number of traces does not guarantee to produce all the relevant behaviours for analysis purposes. As a result, we decided to work on the debugging of LTS models, which represent in a finite way all possible behaviours of the system.

Distance Metrics A relevant line of works on counterexample interpretation is the one by Groce et al. which makes use of *distance metrics.* A distance metric is a function between two executions of the same program. In his first work [Gro04] Groce proposes the use of distance metrics to understand counterexamples from bugged C programs and to determine causal dependencies, exploiting the Lewis counterfactual reasoning to define causality. The cause of the bad behaviour is identified by the delta between the counterexample and the most similar correct execution, which is measured by a given distance metric. Note that, w.r.t. the line of work of Gössler et al., that also deals with causality analysis, this work has a white box approach, where the user is aware of the source code, while Gössler et al. treat software components as black boxes. This approach has been implemented in a tool called *explain* detailed in [GKL04], which makes use of the CBMC model checker to verify ANSI C programs. The work of Groce is extended in [CGS04] with the use of predicate abstraction, a state-space reduction technique that provides an high-level description of the state-space, in conjunction with the MAGIC model checker [CCG$^+$04]. According to the authors, the abstract state-space allows making explanations more informative. The work of Groce is finally improved in [GK05, GCKS06] in order to ease the comprehension of counterexamples. This is done by reducing the counterexample length and automatically hypothesizing causes of the error in [GK05], and by introducing a slicing method to remove irrelevant changes in values between the correct execution and the counterexample in [GCKS06]. Note that the line of works of Groce [Gro04, GKL04, GK05, GCKS06] do not focus on the order of actions in the counterexample, like we and others do (e.g., [LB13]). Instead, they take into account the values of variables that cause the bug. Moreover these techniques only consider single counterexamples. While on one hand this allows better performances, on the other hand taking into account the whole model (as we do) gives a global view of the bug behaviour, allowing the developer to take into account all the situations in which the failure arises.

Minimal Code Changes Griesmayer et al.'s work [GSB07, GSB10] propose to perform minimal changes in a C program so that the counterexample will not fail in order to localize faults. The authors take inspiration from model-based diagnosis to find components (sets of expressions in a given program) that can be changed, in such a way to remove differences between the actual and the intended behaviour. To do this their approach first exploits a BMC model checker to produce a counterexample. Second, it looks for minimal changes in the program that make a component behave in an alternative way, in order to produce a correct execution with the same program inputs. In [dSACdLF17] Alves et al. extend the work of Griesmayer focusing on concurrent programs. To do this, the authors perform code transformations to convert the concurrent original version of the program into a non-deterministic sequential one. The program is then instrumented following [GSB07]. Bounded model checking is finally applied to the sequential version of the program. Note that in our work we do not focus on a specific language, as the works in [GSB07, GSB10, dSACdLF17] do with the C programming language, but we reason on LTS behavioural models, thus remaining generic w.r.t specification and programming languages. Moreover, to find the faulty component they need to modify the code, whereas in our approach we do not modify the model.

Causal Analysis of Counterexamples Two works explicitly take into account causality notions to give a counterexample explanation. The first one [WYIG06], of Wang et al., aims to locate defective program portions and to understand how these cause the failure. The proposed approach consists in a weakest precondition computation algorithm that produces a proof of infeasibility for an input counterexample, in the form of a minimal set of predicates that contradict with each other. Their approach allows focusing on a single counterexample, without the need to compare it to successful executions, thus resulting more scalable w.r.t other works. Causal analysis is also used by Beer et al. in [BBC$^+$12]. The authors adopt the Halpern and Pearl model to explain counterexample in LTL model checking with a focus on hardware systems. Their approach takes as input a single counterexample, in the form of a matrix of variable values. It then exploits an algorithm they built to compute the set of causes and allows the user to focus only on the points of the counterexample which are relevant for the failure. Similarly to the line of work of Groce about distance metrics, [WYIG06, BBC$^+$12] do not focus on the order of actions in the counterexample (as we do), but reason on the values of variables that cause the bug in a single counterexample.

Component Traces Analysis Gössler et al. exploit causality analysis in the interpretation of traces in component-based systems, in order to establish liabilities in case of fault occurrence. In [GMR10] the authors reason about logical causality between contract violations, where a *contract* is used to specify the expected behaviour of a component. The authors rely on the notion of precedence by Lamport [Lam78] to define temporal causality, and introduce three variants of logical causality (necessary, sufficient and horizontal causality). This allows tracing the propagation of faults and establishing whether a prefix of a local trace is part of the cause for the global property violation. In [GM15] Gössler and Le Métayer improve the precision of the previous work by introducing the detection of unaffected prefixes, which are prefixes of component traces that are not affected by the failures. Unaffected prefixes allow distinguishing dependencies between events in component traces and analysing the propagation of failures. Finally, in [WAK$^+$13, WGG$^+$15, GA14] some extensions of Gössler's work for fault diagnosis are proposed for real-time systems. Note that the line of work of Gössler has a focus on component liabilities in component-based systems, while in our case we focus on locating choices in a behavioural model that affect the property satisfaction.

Causality Checking In [LL13, LL14] Leitner-Fischer and Leue propose a causality checking based on the Halpern and Pearl counterfactual reasoning. Causality checking aims at providing insights on the causes of a property violation in model checking and allows the developer to identify sequences of events that bring the system to the violation of the property. This approach considers both the order of events and the non-occurrence of events as causal for a failure. In [LL13] the authors built causality checking as an on-the-fly approach integrated with algorithms for complete state exploration in explicit-state model checking. Their method takes as input concurrent systems described by transition systems, and is implemented using the SpinJa toolset, a Java re-implementation of the SPIN [Hol97] model checker. The approach has been implemented in a tool called SpinCause. The tool has been extended in [LL14] by introducing causality checking of PRISM [KNP02] models and computation of probabilities of combination of causal events. In [BHK$^+$15] the approach proposed by Leitner-Fischer and Leue is improved using a symbolic representation of the state space and SAT-based bounded model checking (in particular, the NuSMV2 model checker [CCG$^+$02]). The adoption of bounded model checking allows constraining the SAT solver in order to generate only error traces with new information about the cause. Comparing to causality checking techniques, we do not have a specific focus on extracting causal relations between events. Our aim is to detect choices in the model between correct and incorrect behaviours, and to use this choices to ease the comprehension of counterexamples.

2.2 Fault Localization Using Testing

Another solution for localization of faults in failing programs consists in using testing techniques, which is quite related to our approach since testing can be seen as a case of non-exhaustive model checking. In this section we will present some examples of testing techniques. The first paragraphs details methods that requires code changes to find the fault, like for instance mutation testing [PT14]. We then present a method that exploits a combination of counterexample analysis and testing in order to find vulnerable program locations and dead code. At the end of the section we discuss pattern mining based techniques (applied on testing), delta debugging based approaches and execution slicing.

Mutation Testing Mutation analysis proposes to introduce changes into the program source code in order to produce mutant programs and to compare their output to the original one in order to expose defects. Le Traon and Papadakis proposed in [PT14] an approach which makes use of a mutation-based fault localization. This paper suggests the use of a sufficient mutant set to locate effectively the faulty statements. Experiments carried out on the well known Siemens test suite [HFGO94] reveal that mutation-based fault localization is significantly more effective than current state-of-the-art fault localization techniques. However, note that the technique proposed in [PT14] require to modify the program code by introducing mutations, while in our case we do not need to modify the code, since we reason only on the program model. Note also that this approach applies on sequential C programs whereas we focus here on models of concurrent programs, without relying on a specific language.

Automatic Correction In [HG04] He and Gupta propose an automated debugging method which combines solutions from testing and weakest precondition used in formal methods. The presented method is able to automatically locate and correct erroneous statement in a faulty function, assuming the correct specification is available in the form of preconditions and postconditions of the function. This approach first detects the faulty statements using a path-based weakest precondition notion, then generates a modification to remove the fault. All the test cases are thus executed again to search for other faulty statements, until all the statements are corrected. Note that faulty programs need to be written in a subset of C, and must be instrumented to generate execution traces. Moreover, the presented approach is only capable to detect at most one error per function, and it is not capable of correcting errors caused by non-terminating loops or segmentation faults.

Target Coverage In [BCH$^+$04] Beyer et al. aim to produce test suites from C programs where the coverage of a given predicate is ensured. To do this, the proposed method extends the BLAST model checker [HJMS02] to first check the reachability of a given target predicate and, second, to use the produced counterexample to find a test vector (initial assignment) that reaches the target predicate. Such an approach is repeated to check whether the control flow graph of the program is completely covered. This approach can be used for finding dead code locations and for discovering security vulnerabilities, for instance finding which parts of a program can be run with root privileges. While our goal is to help the developer in the debugging of safety and liveness properties, the target of Beyer et al. is the detection of C programs vulnerabilities.

Pattern Mining of Test Cases In [FLS06] Di Fatta et al. exploit pattern mining to detect faults in software by analysing test cases. Their approach uses tests cases in the form of function call trees as input and is divided into three main steps. The first step collects and classifies execution traces into two categories, corresponding to correct and failing runs, with the use of an oracle. The second step corresponds to a filtering phase in which frequent pattern mining is performed, using the FREQT [AKA$^+$02, Zak02] pattern mining algorithm, to find frequent subtrees in successful and failing tests executions. The third step represents the analysis phase, in which functions are ranked according to likelihood to contain a bug with the use of ranking methods. As for other works in the testing field, this approach is also evaluated on the Siemens test suite. It is worth mentioning that authors in [FLS06] define a notion of neighbourhood, which correspond to a section of the control flow of a program. In particular, the filtering phase of their approach allows to retrieve the set of discriminative neighbourhoods in the function call trees that are useful in discovering the faults. Even if this notion of neighbourhood has similar purposes to ours (both helps in finding the faults), the two notions are different: their neighbourhood represents a section of the fault call tree, while our neighbourhood notion is a set of incoming and outgoing transitions representing a choice in an LTS model that has an impact on the property compliance. Moreover, the use of testing techniques as the authors do in [FLS06] often requires a high amount of test cases in order to ensure high results quality. In [BFT06] Baudry et al. try to find a solution to this problem by producing a minimum amount of test cases while maximizing the accuracy in diagnosis. Another work which deals with the generation of test cases is the one detailed in [EBNS13], where Elmas et al. present Concurrit, a domain specific language for reproducing concurrency bugs which helps in generating tests. However, relying on a model as we do allows us to avoid the generation of high amounts of test cases.

Delta Debugging A relevant and well known line of work about fault localization using testing techniques is the one of Zeller et al. about *delta debugging* [Zel09]. The aim of delta debugging consists in understanding the minimal differences (deltas) between a successful and a failing program execution. In [Zel99] Zeller first presents delta debugging, implemented as an algorithm to determine code changes between different releases of the same program that cause unexpected behaviours. In [ZH02] Zeller and Hildebrandt apply delta debugging to test case minimization, focusing on understanding which are the characteristics of a test cases that are responsible for a given failure. In [CZ02] Choi and Zeller use delta debugging to discover deltas in thread scheduling that produce concurrency failures. To do this they first generate new runs with alternative thread schedules of a Java program with the DEJAVU tool [CAN$^+$01]. Delta debugging is then used to extract the location in thread schedules where the switch between threads produce a failure. The delta debugging approach is used to locate and isolate cause-effect chains in [Zel02] and [CZ05], by focusing on program's states differences between a successful execution and a failing one. In particular, in [Zel02] the authors have a focus on searching in space, that consists in looking for values and variables that are causal for a failure. In [CZ05] Cleve and Zeller improve this approach by focusing on "cause transitions", which represent program points where a failure cause originates (for instance, for the change of value in a variable). This approach has been implemented in the Igor tool [igo] and evaluated on the Siemens test suite [HFGO94], which consists of various C programs containing each one a defect. These works which rely on delta debugging exhibit some drawbacks. First of all, in [Zel02] the used method requires the developer to introduce instrumentation points on the analysed program, and the validity of execution traces is not guaranteed [GV03]. In some cases (e.g., in [CZ05]) the obtained results cannot be generalized to arbitrary programs. In general, delta debugging has been mainly applied to sequential programs, while our approach has a focus on concurrent programs. Nevertheless Choi et Zeller in [CZ02] focus also on parallel programming, but only from the point of view of thread scheduling. Moreover, test-based approaches cannot verify a given property for all the executions of the program (as the authors of [CZ02] state presenting their work).

Execution Slicing Another relevant debugging method consists in program slicing [Wei82, Tip95]. Program slicing is detailed as the computation of the set of program slices (subsets of program statements) that might affect given variables causing the fault, allowing the extraction of accurate data dependencies between variables which influence the fault. Slicing techniques comprehend static slicing, originally proposed by Weiser [Wei79], and dynamic slicing, which applies to a specific program run [AH90, KL90]. A particular slicing technique, called execution slicing [AHLW95], adopts a testing approach. Execution slicing exploits test cases to infer faulty statements. It first computes for each test case a set of corresponding executed statements (a slice). Second, it extracts the difference between slices in failing and correct test cases in order to detect the faulty statements. Reasoning on test cases, as it is done in execution slicing, brings the typical drawbacks of testing techniques, for instance the need to generate an high amount of tests cases.

2.3 Bug Visualization

It has always been important to give a visual feedback to the developer in debugging techniques. In [Lie97] Lieberman, while presenting the work of Baecker et al. about software visualization for debugging [BDM97], said that "to debug a program, you have to see what it is doing". We present here various examples of visualization techniques applied in fault debugging.

Code Colouring Various tools aim at performing code colouring to support verification techniques. As an example, the work presented by Jones et al. [JHS02] aims at helping the user in debugging faults, using visualization to assist the user in finding statements that might contain faults. To do this, it colours program statements in order to distinguish between the statements executed only in faulty executions and the ones executed only in successful runs. Colouring is also used in the Maude [CDE$^+$07] system for debugging Maude programs. If a term in a program does not fully reduce, term colouring is used to better understand the problem. Colouring techniques are also used for bug visualization in static analysis tools, such as Polyspace [Pol]. Polyspace is a product built by Mathworks that exploits static code analysis to discover critical runtime errors. It allows the visualization of the control flow using the call hierarchy and the call flow graphs. Since we reason on the LTS, we do not provide any code colouring. However, we provide transition colouring in the visualized LTS, highlighting different types of transitions. Colouring the specification that generated the LTS could be taken into account by using structural information, as we discuss in Chapter 9.

Visualization of Traces and Models Trace visualization methods have been provided in some of the approaches we presented in previous sections. For instance, in [BBC$^+$12] trace visual explanations are shown in the form of red dots exploiting the IBM RuleBase PE trace viewer. The SpinCause tool proposed by [LL14] provides visualization of causal relationships with fault trees, which are derived from safety and reliability engineering and are used to map dependency relationship between faulty events. Note that various model checkers are provided with basic visualizer components for visualizing models. As an example, the CADP tool [GLMS13] offers a 2D component to draw and edit interactively a PostScript representation of a BCG graph. The Uppaal model checker [LPY97] also provides a graphical simulator to visualize the dynamic behaviours of a system. However these components only give a global view of the system, without pointing out system parts affected by a bug.

Visualization of Very Large State Spaces An important part of our method relies on bug visualization techniques. The closest work to ours is a tool for visualizing the structure of very large state spaces developed by Groote and Van Ham [GvH03, GvH06]. This approach relies on a clustering method to generate a simplified representation of the state space, and can be useful for better understanding the overall structure of the model. To do this, this method builds a 3D representation in the form of a tree, which constitutes the backbone of the whole graph. The rest of the tree structure is built by clustering sets of states. The use of clustering techniques provides an high scalability to the approach, since individual states and transitions are not displayed but are grouped in sets. This tool is particularly useful for better understanding the overall structure of the model, but this work does not necessarily target debugging, which is our main objective here. Indeed, in contrast, we do not have only as input a specification and its corresponding model, but also a temporal property. This property allows us to distinguish correct and incorrect behaviours in our model, and our visualization techniques focus on this question in order to identify a particular structure that would help the developer to understand and identify the source of the bug.

2.4 Alternative Approaches

Some relevant debugging methods do not follow a trace-based nor a testing approach. This is the case of techniques that perform analysis on the code (e.g., [SY15]), or on control flow graphs (e.g., [BCR15]).

Coverage Analysis In [SY15], the authors propose a new approach for debugging value-passing process algebra through coverage analysis. The authors define several coverage notions before showing how to instrument the specification without affecting original behaviours. This approach helps one to find errors such as ill-formed decisions or dead code, but does not help to understand why a property is violated during analysis using model checking techniques.

Flow Driven Approaches LocFaults [BCR15] is a flow-driven and constraint-based approach for error localization. This work focuses on programs with numerical statements and relies on a constraint programming framework allowing the combination of Boolean and numerical constraints. It takes as input a faulty C program for which a counterexample and a postcondition are provided. This approach makes use of constraint based bounded model checking combined with a minimal correction set (MCS) notion to locate errors in the faulty program. A MCS represents the set of constraints that must be removed to make the constraint system satisfiable. Their approach first builds a control flow graph (CFG) from the program. Second, it generates a constraint system for the paths in the CFG. Finally, MCSs are computed for each path in the CFG. CPBPV [CRH10] is used as constraint based model checker, allowing building both the CFG and generating the constraint system on-the-fly. Note that authors do not mention whether their solution could be used for analysing concurrent programs.

2.5 Concluding Remarks

Our approach can be placed among the trace-based approaches. In particular, the line of research which focuses on interpreting counterexample and favouring their comprehension (e.g., [JRS02, GV03, BNR03]) is the closest to our work.

Reasoning on a behavioural model as we do brings various advantages. First of all, some of the approaches we reviewed do not have a specific focus on concurrent systems. This is the case of [RS04, ESW12] among trace-based techniques and of [PT14, BCH$^+$04, HG04] among testing ones. In our case our LTS based approach allows handling both sequential and concurrent systems. Various works need to focus on a specific language, like [dSACdLF17, GSB07, GSB10, BCH$^+$04] which have a focus on C programs, and need to modify or instruments code, as in [PT14, HG04, Zel02]. Instead, reasoning on LTS behavioural models allows us to be generic w.r.t. specifications and programming languages. Some existing approaches rely on analysis of traces, as [BWW14, LB12, LB13]. However, reasoning on traces do not allows handling all the executions, while reasoning on the LTS, which represents all the behaviour of a system, allows us to solve this issue.

Chapter 3

Preliminaries

This chapter presents some preliminary notions that we use in the rest of this work. We first introduce Labelled Transition Systems. Then, the LNT specification language is described, followed by the presentation of temporal properties. At the end of the chapter we present some operations, definitions and algorithms that we use on LTSs.

3.1 Models

In this thesis we adopt *Labelled Transition System (LTS)* as behavioural model of concurrent programs. LTSs are used by a large set of concurrent system verification tools, such as CADP [GLMS13], mCRL2 [CGK+13], LTSmin [KLM+15] and TAPAs [CDLT08]. An LTS consists of states and labelled transitions connecting these states.

Definition 1 *(LTS) An LTS is a tuple $M = (S, s^0, \Sigma, T)$ where S is a finite set of states; $s^0 \in S$ is the initial state; Σ is a finite set of labels; $T \subseteq S \times \Sigma \times S$ is a finite set of transitions.*

A transition is represented as $s \xrightarrow{l} s' \in T$, where $l \in \Sigma$. An LTS is usually produced from a higher-level specification of the system described with a process algebra for instance. Specifications can be compiled into an LTS using specific compilers.

An LTS can be viewed as all possible executions of a system. One specific execution is called a *trace*.

Definition 2 *(Trace) Given an LTS $M = (S, s^0, \Sigma, T)$, a trace of size $n \in \mathbb{N}$ is a sequence of labels $l_1, l_2, \ldots, l_n \in \Sigma$ such that $s^0 \xrightarrow{l_1} s_1 \in T, s_1 \xrightarrow{l_2} s_2 \in \; T, \ldots, s_{n-1} \xrightarrow{l_n} s_n \in T$. A trace is either infinite because of loops or the last state s_n has no outgoing transitions. The set of all traces of M is written as $t(M)$.*

Note that $t(M)$ is prefix closed. One may not be interested in all traces of an LTS, but only in a subset of them. To this aim, we introduce a particular label δ, called *final label*, which marks the end of a trace, similarly to the notion of accepting state in language automata. This leads to the concept of *final trace.*

Definition 3 *(Final Trace) Given an LTS $M = (S, s^0, \Sigma, T)$, and a label δ, called final label, a final trace is a trace $l_1, l_2, \ldots, l_n \in \Sigma$ such that $s^0 \xrightarrow{l_1} s_1 \in T, s_1 \xrightarrow{l_2} s_2 \in T, \ldots, s_{n-1} \xrightarrow{l_n} s_n \in T$, $l_1, l_2, \ldots, l_n \neq \delta$ and there exists a final transition $s_n \xrightarrow{\delta} s_{n+1}$. The set of final traces of M is written as $t_\delta(M)$.*

Note that the final transition characterized by δ does not occur in the final traces and that $t_\delta(M) \subseteq t(M)$. Moreover, if M has no final label then $t_\delta(M) = \emptyset$.

In this thesis, we use LNT [CCG+18] as specification language and compilers from the CADP toolbox [GLMS13] for obtaining LTSs from these specifications. We present in the next section a short introduction to the LNT language, in order to let the reader understand some LNT excerpts presented in Section 8. It is worth noting that, although we rely on LNT in this thesis, our approach is generic in the sense that it applies on LTSs produced from any specification language and any compiler/verification tool.

3.2 LNT

LNT is an improved version of LOTOS [BB87], providing a more user-friendly notation which takes inspiration from typical imperative and functional programming languages notations. LNT allows the definition of data types, functions, and processes, which we present in the next paragraphs. The reader interested in more details about LNT should refer to [CCG+18]. Note that comments in LNT can be single lines comments, in the form "- - comment ", or block comments, in the form "(* comment *)". LNT keywords are highlighted in bold.

3.2.1 Data Types

LNT provides some predefined types, i.e., Boolean (**bool**), natural numbers (**nat**), integer numbers (**int**), real numbers (**real**), characters (**char**) and strings (**string**). The user can also declare abstract data types, through the use of the **type** keyword.

3.2.2 Functions

Functions are defined with the **function** keyword. A function can have zero, one or more arguments and returns a result. The predefined data types are already provided with predefined functions. Various statements can be used inside a function, for instance correct termination (**null**), sequential composition (**;**), function return (**return**), variable declaration (**var**), conditional construct (**if**), breakable loop construct (**loop**), for loop construct (**for**). Value assignments to variables is performed with the **:=** operator. Table 3.1 provides a simplified overview of function statements syntax where I stands for a statement, V stands for an expression, x for a variable identifier, T stands for a type identifier and L is a loop label.

Table 3.1: LNT function statements syntax.

$$
\begin{array}{rcl}
I & ::= & \textbf{null} \\
 & | & I_1; I_2 \\
 & | & return[V] \\
 & | & \textbf{var } x{:}T \textbf{ in } x := V; I_0 \textbf{ end var} \\
 & | & \textbf{if } V \textbf{ then } I_0 \textbf{ end if} \\
 & | & \textbf{loop } L \textbf{ in } I_0 \textbf{ end loop} \\
 & | & \textbf{for } I_0 \textbf{ while } V \textbf{ by } I_1 \textbf{ loop } I_2 \textbf{ end loop}
\end{array}
$$

3.2.3 Processes

Processes in LNT are defined with the **process** keyword. As for functions, they have zero, one or more arguments, but they do not return a result. Statements presented for functions in Table 3.1 also apply to processes. Moreover, LNT processes are also built from behaviours, e.g. actions, nondeterministic choices (**select**), parallel composition (**par**) and process instantiation. A process terminates by executing implicitly the special action **exit**. In order to manage communications and synchronizations between processes, a list of gates where they can perform actions is provided (between square brackets).

Gates, Offers and Channels

A gate is used by an LNT process to handle input/output communications and synchronizations. A gate is defined as $G(O_1, \ldots, O_n)$, where G stands for a gate and O stands for an offer. Offers allows to exchange data on a gate and they can be an emission of an expression or the reception of a value. In the first case an offer is highlighted by "!V", where V stands for an expression. Note that the "!" is optional. In the second case the offer must be highlighted by "?x", where x stands for a variable identifier which stores the received value.

Gates can be untyped (**none**) or typed by a channel. In this latter case the channel defines the gate profile, thus limiting offers profiles accepted by a gate. A channel is defined using the **channel** keyword and must state one or more profiles which are lists of parameters that declare the number and types of authorized offers. Note that two gates are compatible only if they have the same type or they are both untyped.

Visible events on a process execution are represented by actions. Actions can be internal ("i" action) or realised on a gate. An action on a gate is defined with the gate identifier. Note that internal actions never have offers.

Nondeterministic Choice

Nondeterministic choices between two or more behaviours can be performed in processes through the **select** operator. The syntax of a nondeterministic choice is the following, where B stands for a behaviour: **select** B_1 [ ]...[ ] B_n **end select**. Note that this corresponds to a state in the LTS with multiple outgoing transitions, where each transition corresponds to one of the B_i in a **select** construct.

Parallel Composition

The parallel composition operator **par** allows to declare multiple behaviours that are executed in parallel. The syntax for the parallel composition is the following, where B stands for a behaviour and G for a gate: **par** $G_1, ..., G_m$ **in** $B_1||...||B_n$ **end par**. The communication between behaviours $B_0, \ldots, B_n$ is carried out by rendezvous on a list of synchronized actions included in the synchronization set $G_1, \ldots, G_m$.

The parallel composition operator can be used to handle communication between different processes using process instantiations as behaviours in the operator. One or more gates can be declared in the synchronization set in order to specify on which gates the processes must synchronize their actions. The ones not specified in the synchronization set are executed independently by each process. Only actions which are offered by every process in the parallel composition can be synchronized. Note that the **exit** action is always synchronized between all processes at the end of a parallel composition.

3.3 Temporal Properties

Model checking consists in verifying that an LTS model satisfies a given temporal property φ, which specifies some expected requirement of the system. Temporal properties are usually divided into two main families: *safety* and *liveness* properties [BK08].

3.3.1 Safety Properties

Safety properties are widely used in the verification of real-world systems. Safety properties state that "*something bad never happens*". A safety property is usually formalised using a temporal logic. To do this, we use Model Checking Language (MCL) [MT08] in Chapter 8. MCL is an action-based, branching-time temporal logic used in order to express concurrent systems temporal properties. A safety property can be semantically characterized by a possibly infinite set of traces t_φ, corresponding to the traces that violate the property φ in an LTS. If the LTS model does not satisfy the property, the model checker returns a *counterexample*, which is one of the traces characterised by t_φ.

Definition 4 *(Counterexample for Safety Properties) Given an LTS $M = (S, s^0, \Sigma, T)$ and a safety property φ, a counterexample is any trace which belongs to $t(M) \cap t_\varphi$.*

3.3.2 Liveness Properties

Liveness properties state that "*something good eventually happens*". We focus on a class of liveness properties, called *inevitable execution* properties. Most of the patterns that commonly occur in the specification of liveness properties make use of the inevitable executions. This is the case of the *Response Property Pattern*, that is the most common pattern in [DAC99]. An inevitable execution property states that, given an LTS M and an action l, every trace from the initial state in M presents a transition with the action l. In this thesis we support *nested inevitable executions.* For instance, a property with two nested actions l_1 and l_2 states that every trace in a given model must exhibit the action l_1 later followed by the action l_2. Note that the two actions do not need to be contiguous in traces. To express nested inevitable executions we define a *nested inevitability operator* using the *Action-based Computation Tree Logic (ACTL)* [DV90], which is another action-based branching-time temporal logic:

Definition 5 *(Nested Inevitability Operator) Given a sequence of labels $l_1, \ldots l_n$, the nested inevitability operator is defined as*

$$\texttt{\textbf{Inev}}(l_1, l_2, \ldots l_n) = A[true_{true}U_{l_1}A[true_{true}U_{l_2} \ldots A[true_{true}U_{l_n}true] \ldots]]$$

where A and U denote the ACTL operators along All paths *and* Until*, resp.*

A nested inevitable execution property can be semantically characterised by a possibly infinite set of traces t_φ, corresponding to the traces that comply with the property φ in an LTS. If the LTS model does not satisfy the property, the model checker returns a *counterexample*, which is one of the traces characterised by $t(M) \setminus t_\varphi$.

Definition 6 *(Counterexample for Liveness Properties) Given an LTS $M = (S, s^0, \Sigma, T)$ and a liveness property φ, a counterexample is any trace which belongs to $t(M) \setminus t_\varphi$. A counterexample can be in the form of an elementary trace, which is a trace where states are* *pairwise distinct, or a lasso, which is a trace* $s^0 \xrightarrow{l_1} s_1 \in T, \ldots, s_{n-2} \xrightarrow{l_{n-1}} s_{n-1} \in T, s_{n-1} \xrightarrow{l_n} s_n \in T$*, such that* $s^0 \xrightarrow{l_1} s_1 \in T, \ldots, s_{n-2} \xrightarrow{l_{n-1}} s_{n-1} \in T$ *is an elementary trace and* $s_n = s_i$ *for some* $0 \leq i < n$*.*

3.4 Operations on LTS

In this section we present some definitions, operations and algorithms which are used in the rest of this work in order to handle LTSs.

3.4.1 Simulation Relation

The simulation relation [Par81] allows to relate two LTSs which behave in the same way. It is defined formally as follows:

Definition 7 *(Simulation Relation) Given two LTSs* $M_1 = (S_1, s_1^0, \Sigma_1, T_1)$ *and* $M_2 = (S_2, s_2^0, \Sigma_2, T_2)$*,* M_1 *is simulated by* M_2*, written* $M_1 \sqsubseteq M_2$*, iff* $\exists R \subseteq S_1 \times S_2$ *such that* $R(s_1^0, s_2^0)$ *and* $\forall s_1 \in S_1, s_2 \in S_2, R(s_1, s_2)$ *implies* $\forall s_1 \xrightarrow{l} s_1' \in S_1, \exists s_2 \xrightarrow{l} s_2' \in S_2$*,* $R(s_1', s_2')$*.*

3.4.2 LTS Determinization

The LTS determinization operation allows the conversion of a non-deterministic LTS to a deterministic one. A deterministic LTS is formally defined as follows:

Definition 8 *(Deterministic LTS) Given an LTS* $M = (S, s^0, \Sigma, T)$*,* M *is deterministic iff* $\forall s \in S$ *and* $\forall l \in \Sigma$ *there exists at most one* $s' \in \Sigma$ *such that* $s \xrightarrow{l} s'$*.*

To determinize an LTS we perform an LTS reduction modulo weak trace equivalence, meaning that the reduced LTS contains all the visible transitions sequences of the original LTS. The determinization is performed with the Reductor tool of the CADP toolbox, which makes use of the classic subset construction algorithm defined in [ASU86]. Thus, given an LTS M, the resulting M' is a deterministic LTS which does not contain any invisible transition and is weakly trace equivalent to M.

3.4.3 LTS Minimization

Given an LTS M, an LTS minimization operation on M consists in transforming M into an equivalent LTS M' which have a minimum number of states. The equivalence between M and M' is guaranteed by the *strong bisimulation* [Mil89] relation.

Definition 9 *(Strong Bisimulation Relation) Given two LTSs* $M_1 = (S_1, s_1^0, \Sigma_1, T_1)$ *and* $M_2 = (S_2, s_2^0, \Sigma_2, T_2)$*,* M_1 *and* M_2 *are strongly bisimilar, written* $M_1 \approx M_2$*, iff* $\exists R \subseteq S_1 \times S_2$ *such that* $R(s_1^0, s_2^0)$ *and* $\forall s_1 \in S_1, s_2 \in S_2, R(s_1, s_2)$ *implies the following two conditions:*

- $\forall s_1 \xrightarrow{l} s_1' \in S_1, \exists s_2 \xrightarrow{l} s_2' \in S_2$ *and* $R(s_1', s_2')$*;*

- $\forall s_2 \xrightarrow{l} s_2' \in S_2, \exists s_1 \xrightarrow{l} s_1' \in S_1$ *and* $R(s_1', s_2')$*.*

To perform the minimization operation we make use of the bcg_min tool provided by the CADP toolbox, which implements sequential variants of the partition refinement algorithm for reducing an LTS modulo strong bisimulation detailed by Blom et Orzan in [BO05].

3.4.4 Synchronous Product

A synchronous product between two LTSs M_1 and M_2, written $M_1||M_2$, is defined as follows:

Definition 10 *(Synchronous Product) Given two LTSs* $M_1 = (S_1, s_1^0, \Sigma_1, T_1)$ *and* $M_2 = (S_2, s_2^0, \Sigma_2, T_2)$ *the synchronous product between* M_1 *and* M_2*, written* $M_1||M_2$*, is an LTS* $M = (S, s^0, \Sigma, T)$ *where* $S = S_1 \times S_2$*;* $s_0 = \langle s_1^0, s_2^0 \rangle$*;* $\Sigma = \Sigma_1 \cup \Sigma_2$*;* T *such that:*

- $\langle s_1, s_2 \rangle \xrightarrow{l} \langle s_1', s_2 \rangle$ *if* $l \in \Sigma_1, l \notin \Sigma_2$ *and* $s_1 \xrightarrow{l} s_1'$*;*

- $\langle s_1, s_2 \rangle \xrightarrow{l} \langle s_1, s_2' \rangle$ *if* $l \notin \Sigma_1, l \in \Sigma_2$ *and* $s_2 \xrightarrow{l} s_2'$*;*

- $\langle s_1, s_2 \rangle \xrightarrow{l} \langle s_1', s_2' \rangle$ *if* $l \in \Sigma_1 \cap \Sigma_2$ *and* $s_1 \xrightarrow{l} s_1'$ *and* $s_2 \xrightarrow{l} s_2'$*.*

3.4.5 Strongly Connected Components

In Chapter 6 we need to split an LTS in smaller portions in order to handle cycles. To do this we use the notion of *Strongly Connected Component (SCC)* [Tar72], that is a partition of an LTS where every state is reachable from any other state.

Definition 11 *(Strongly Connected Component (SCC) in LTS) Given an LTS* $M = (S, s^0, \Sigma, T)$*, an SCC of* M *is a tuple* $G = (S_G, \Sigma_G, T_G)$ *where:*

- $S_G \subseteq S$ *is a finite set of states;*

- $\Sigma_G \subseteq \Sigma$ *is a finite set of labels;*

- $T_G \subseteq S_G \times \Sigma_G \times S_G, T_G \subseteq T$ *is a finite set of transitions;*

- *for every pair of states* $(s', s'') \in S_G$ *there exists a path from* s' *to* s'' *such that* $s' \xrightarrow{l} s_1 \in T_G, s_1 \xrightarrow{l} s_2 \in T_G, \ldots, s_{n-1} \xrightarrow{l} s'' \in T_G$ *and there exists a path from* s'' *to* s' *such that* $s'' \xrightarrow{l} s_1 \in T_G, s_1 \xrightarrow{l} s_2 \in T_G, \ldots, s_{n-1} \xrightarrow{l} s' \in T_G$*.*

Note that s and s' in Definition 11 can be the same, thus allowing also SCCs of only one element. Moreover every SCC in an LTS is also a Maximally Strongly Connected Component, since an SCC cannot be subsumed by a larger SCC by definition.

Tarjan Algorithm

To detect all the SCCs in an LTS we use the Tarjan's SCCs algorithm [Tar72]. Given an LTS $M = (S, s^0, \Sigma, T)$, the algorithm allows the detection of all the SCCs in linear time, with a cost of $O(|S| + |T|)$. The algorithm makes use of a modified depth-first search procedure. The pseudo-code is given in Algorithm 1. INDEX and LOWLINK are two map data structures where the key is a state and the value is an integer value. INDEX allows to index states in the order they are reached during the exploration of the LTS. LOWLINK allows to keep the index of the smallest state in the same strongly connected component. For instance, given a state s, LOWLINK(s) is the smallest vertex in the same component as s. The stack *Stack* collects the states that have been already reached but that have not yet been assigned to an SCC. Functions PUSH and POP performs the classic element addition and removal operations on the stack. The loop at line 6 calls the STRONGCONNECT function on all the states that have not been numbered yet

The STRONGCONNECT function represents the core element of the Tarjan's algorithm. The loop at line 16 allows iterating over state s successors (retrieved with the GETSUCCESSORS function), exploring with recursion on STRONGCONNECT the states that have not yet been discovered and updating LOWLINK(s). MIN is a function that compute the minimum between two values. Finally the if statement at line 22 allows returning an SCC when the s state is identified as a root state of the SCC. At the end of the execution of the loop at line 6 all the SCCs in M are identified and retrieved.

Algorithm 1 Tarjan SCC Algorithm

```
procedure TARJAN($M$)
    $i \leftarrow 0$
    INDEX($s$) $\leftarrow \varnothing$
    LOWLINK($s$) $\leftarrow \varnothing$
    $Stack \leftarrow \varnothing$
    for all $s \in S_M$ do
        if $s \notin$INDEX then
            $G \leftarrow$ STRONGCONNECT(s)
            add $G$ to the set of discovered SCCs

    function STRONGCONNECT($s$)
        $i \leftarrow i + 1$
        INDEX($s$) $\leftarrow i$
        LOWLINK($s$) $\leftarrow i$
        PUSH($Stack_G$, $s$)
        for all $s' \in$ GETSUCCESSORS($s$) do
            if $s' \notin$INDEX then
                STRONGCONNECT($s'$)
                LOWLINK($s$) $\leftarrow$ MIN(LOWLINK($s$),LOWLINK($s'$))
            else if INDEX($s'$) $<$ INDEX($s$) then
                LOWLINK($s$) $\leftarrow$ MIN(LOWLINK($s$),INDEX($s'$))
        if LOWLINK($s$) $=$ INDEX($s$) then
            $G$ is a new SCC
            do
                $w \leftarrow$POP($Stack_G$)
                add state $w$ to $S_G$
            while $w \neq s$
        return $G$
```

Chapter 4

Approach Overview

The goal of our work is to simplify the comprehension of counterexamples by taking into account relevant actions that are of prime importance to understand the bug. More precisely, our approach aims at highlighting choices in the model that contains such relevant actions and that can thus explain the violation of a property. These choices are important from a debugging point of view, since they represent decision points where the specification goes from a (potentially) correct behaviour to an incorrect one, thus they can provide an explanation of the bug. We call such choices *neighbourhoods*.

To detect neighbourhoods, our approach takes as input an LTS and a property. Then it first performs a transition categorization step in which transitions in the LTS are typed according to the behaviour of the system they lead to. Three types of transitions are recognized: *correct transitions*, which identify correct behaviours; *incorrect transitions*, which identify incorrect behaviours and *neutral transitions*, which are common to both correct and incorrect behaviours. The LTS where all the transitions have been categorized is called *tagged LTS* and allows us to identify neighbourhoods. The neighbourhood detection step takes as input the tagged LTS and analyses each state of the LTS. States where at least one of the incoming transition is a neutral one and at least one of the outgoing transitions is not neutral are identified as neighbourhoods. The set of recognized neighbourhoods can finally be exploited by the user with visual and abstraction techniques. The visual rendering allows the developer to have a global view of the bug behaviour. Abstractions techniques instead allow building simplified versions of counterexamples, for instance keeping only actions that belong to neighbourhoods. These main steps of our approach are summarised in Figure 4.1.

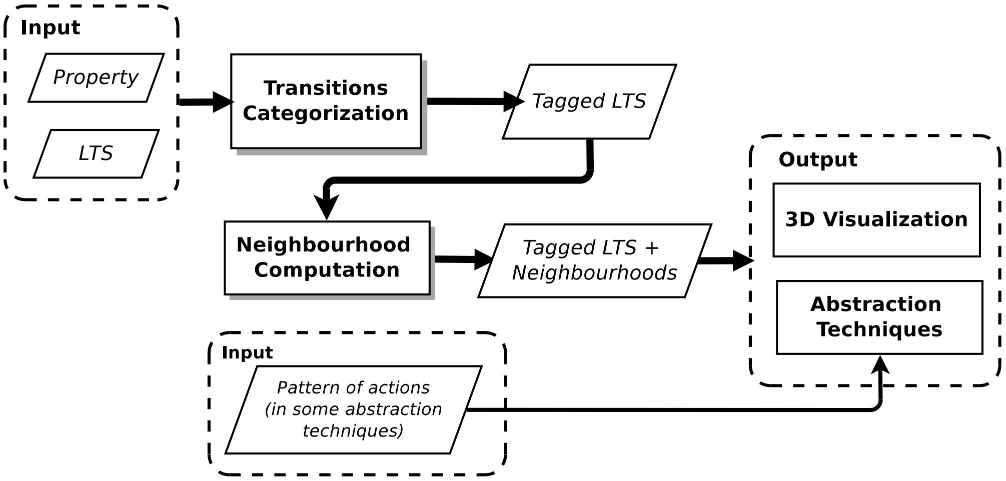

Input
Property
LTS
Transitions Categorization
Tagged LTS
Neighbourhood Computation
Tagged LTS + Neighbourhoods
Output
3D Visualization
Abstraction Techniques
Input
Pattern of actions (in some abstraction techniques)

Figure 4.1: Overview of our approach.

In this chapter we first present the transitions types, we define the neighbourhood notion and we provide a neighbourhood taxonomy. We then propose a methodology to exploit neighbourhoods, using the visualization and the abstraction techniques.

Note that in order to recognize transition types in a model we have built two different approaches: one for handling safety properties, called *Counterexample LTS* approach, and another one for liveness properties, called *Prefix/Suffix* approach. We describe precisely these two approaches in Chapter 5 and in Chapter 6. In this chapter we focus on how the transition types, recognized with these two approaches, are exploited to identify neighbourhoods.

4.1 Transitions Types and Tagged LTS

We are interested in detecting relevant choices in an LTS model between different behaviours. Such behaviours can be correct, meaning that they satisfy a given property, or incorrect, meaning that they violate the given property. Relevant choices are characterized in a model by states with different types of outgoing transitions, meaning that some transitions belong to correct behaviours while some others to incorrect ones. In order to detect the choices in which we are interested, we first need to categorize the transitions in the model into different types. The transition type allows to highlight the compliance with the property of the paths in the model that traverse that given transition. Transitions in the LTS can be categorized into three types:

- **correct transitions**, which belong to paths in the model that represent behaviours which always *satisfy* the property.

- **incorrect transitions**, which belong to paths in the model that represent behaviours which always *violate* the property.
- **neutral transitions**, which belong to portions of paths in the model which are common to correct and incorrect behaviours.

We add the information concerning the detected transitions type (correct, incorrect and neutral transitions) to the LTS in the form of tags. We define the set of transition tags as $\Gamma = \{correct, incorrect, neutral\}$. Given an LTS $M = (S, s^0, \Sigma, T)$, a tagged transition is represented as $s \xrightarrow{(l,\gamma)} s'$, where $s, s' \in S$, $l \in \Sigma$ and $\gamma \in \Gamma$. Thus, an LTS in which each transition has been tagged with a type is called *tagged LTS*.

Definition 12 *(Tagged LTS) Given an LTS $M = (S, s^0, \Sigma, T)$, and the set of transition tags Γ, the tagged LTS is a tuple $M_T = (S_T, s_T^0, \Sigma_T, T_T)$ where $S_T = S$, $s_T^0 = s^0$, $\Sigma_T = \Sigma$, and $T_T \subseteq S_T \times \Sigma_T \times \Gamma \times S_T$.*

Figure 4.2 depicts on the left hand side a portion of an LTS and on the right hand side the same portion in the corresponding tagged LTS where transitions types have been recognized. Correct transitions are depicted with black lines, incorrect ones are depicted with grey dotted lines and neutral ones are depicted with black dotted lines. Correct transitions, represented by the transitions with *QPut* and *Rack* actions, belong to paths that satisfy a given property. On the contrary, the incorrect transition with the *Send* action belong to paths that never satisfies the given property. The neutral transitions with the *Recv* represent a behaviour that is common to both correct and incorrect behaviours.

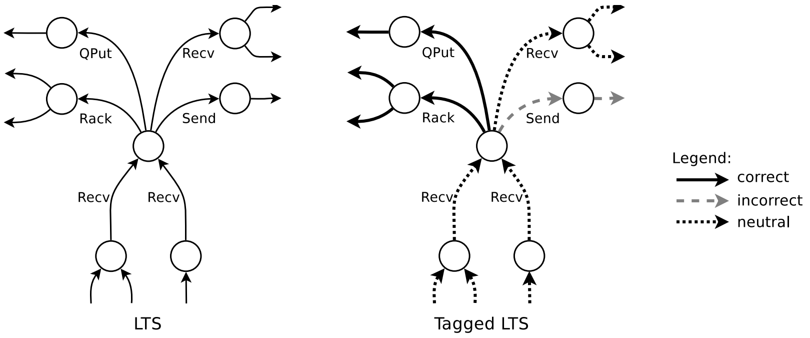


Figure 4.2: Example of Tagged LTS.

4.2 The Neighbourhood Notion

The tagged LTS where transitions have been typed allows us to identify *neighbourhoods* in states in which an incoming neutral transition is followed by a correct or an incorrect one. A neighbourhood represents a choice in the model between two (or more) different behaviours, and consists of all the neutral incoming transitions and all the outgoing transitions. This set of transitions identifies specific parts of the specification that may explain the appearance of the bug and are therefore meaningful from a debugging perspective.

For instance, let us suppose a model of a concurrent system and a simple safety property which states that we never want a given action A after an action B. Let us also suppose that this property is false, since the system executes in some cases an action A after an action B. In order to produce such incorrect cases, the system execution takes some precise choices. Such choices can correspond to some specific interleaving generated by a parallel composition operator or to nondeterministic choices in the specification from which the LTS has been produced. Our neighbourhood notion allows us to identify these important decision points in the model.

Note that only neutral transitions are taken into account as incoming transitions for the neighbourhood. The incoming neutral transitions in a neighbourhood are called *relevant predecessors*, since they highlight actions performed just before the ones described by the correct (or incorrect) transitions. Relevant predecessors are always neutral transitions, since they represent common prefixes for correct and incorrect transitions in neighbourhoods. These transitions are important from a debugging perspective since they represent the last action that can be executed without an impact on the choice represented by the neighbourhood. Thus, they are useful to the developer in order to locate the cause of the bug in the specification.

Definition 13 *(Neighbourhood) Given the tagged LTS $M_T = (S_T, s_T^0, \Sigma_T, T_T)$, a state $s \in S_T$, such that $\exists t = s' \xrightarrow{(l,\gamma)} s \in T_T$, t is a neutral transition, and $\exists t' = s \xrightarrow{(l,\gamma)} s'' \in T_T$, t' is a correct or an incorrect transition, the neighbourhood of state s is the set of transitions $T_{nb} \subseteq T_T$ such that for each $t'' \in T_{nb}$, either $t'' = s' \xrightarrow{(l,\gamma)} s \in T_T$ or $t'' = s \xrightarrow{(l,\gamma)} s''' \in T_T$.*

We now illustrate the example of a portion of a Tagged LTS depicted in Figure 4.2. The incoming and outgoing transitions for the state highlighted in grey in the tagged LTS in Figure 4.3 correspond to a neighbourhood. Note that the depicted state presents a choice between different behaviours, highlighted by the different transition types. This neighbourhood allows us to understand that the choice of the outgoing transition is relevant from a debugging perspective, since it affects the property compliance. For instance, we can see that if we choose a *Rack* action after the *Recv* action, the given property will be satisfied. On the contrary, choosing the *Send* action after the *Recv* one will lead to an incorrect behaviour, thus a property violation.

4.2.1 Neighbourhood Taxonomy

In order to detect all the neighbourhoods in the model, we built a simple procedure which explores all the states in the tagged LTS and checks their incoming and outgoing transitions. Given a state s, if there exists at least one incoming transition in s that is neutral, and at least one of the outgoing transitions of s is not neutral, then s is a neighbourhood. Note

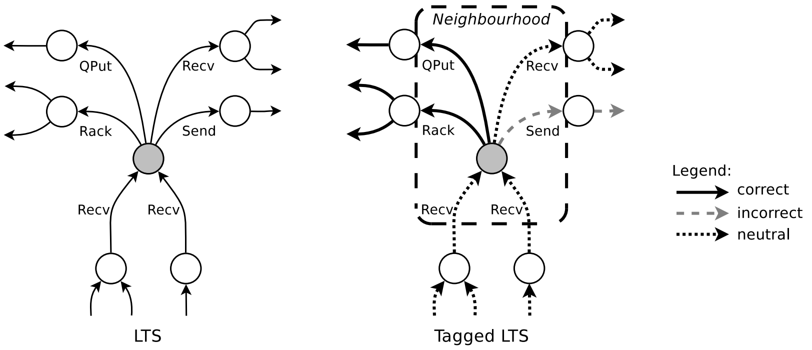


Figure 4.3: Example of neighbourhood with correct transitions.

that there exists a particular case of neighbourhood which does not require an incoming neutral transitions. This is the case of a neighbourhood in the initial state of the LTS.

Given a state *s* in which a neighbourhood has been detected, we can identify a neighbourhood type by looking at its outgoing transitions. More precisely, we can categorise neighbourhoods in four types (as depicted in Figure 4.4 from left to right):

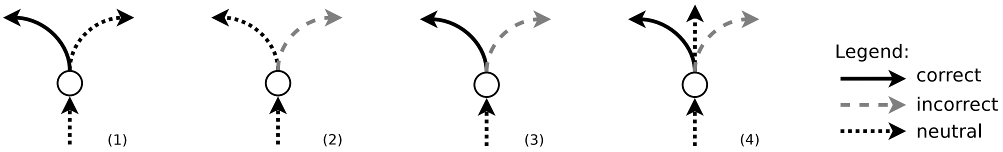


Figure 4.4: The four types of neighbourhoods.

1) with at least one *correct transition* (and no *incorrect transition*). The transitions contained in this type of neighbourhood highlight a choice that can lead to behaviours that always satisfy the property. Note that neighbourhoods with only correct outgoing transitions are not possible, since they would not highlight such a choice. Consequently, this type of neighbourhood always presents at least one outgoing *neutral transition*.

2) with at least one *incorrect transition* (and no *correct transition*). The transitions contained in this type of neighbourhood highlight a choice that can lead to behaviours that always violate the property. Figure 4.5 shows a piece of a tagged LTS where the state in the centre of the figure represents the origin of sequences of incorrect transitions. Note that while a neutral outgoing transition is usually present to highlight the choice, a particular case where only incorrect outgoing transitions are exposed exists. This is the case in which the property is always false and the neighbourhood is located at the initial state of the tagged LTS.

3) with at least one *correct transition* and at least one *incorrect transition*, but no *neutral transition*. This type of neighbourhood highlights a choice between a correct and an incorrect behaviour.

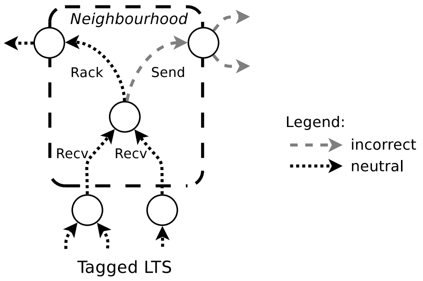


Figure 4.5: Example of neighbourhood with an incorrect transition.

4) with at least one *correct transition*, at least one *incorrect transition* and at least one *neutral transition*. As for neighbourhood of type 3), this type also highlights a choice between a correct and an incorrect behaviour, but the presence of the neutral transition also allows us to choose a behaviour whose correctness is still not known at this point.

4.3 Neighbourhood Exploitation

The tagged LTS where neighbourhoods have been discovered can finally be exploited to extract precise information related to the causes of the bug. To do this, the developer can use two different methods: visualization techniques and abstraction techniques (see Figure 4.6). The visualization techniques rely on a 3D visual rendering of the tagged LTS with neighbourhood, and provide functionalities to facilitate the manipulation of those models (e.g., step-by-step path traversal, counterexample visualization, etc.). The abstraction techniques aim at removing irrelevant parts of the counterexample and highlighting relevant ones to simplify its comprehension. We now define a debugging procedure which relies on the visualization and the abstraction techniques, and we successively present in detail these two methods. The debugging procedure we propose provides hints to the developer to discover the source of the bug and thus favours the comprehension of its cause, by exploiting the notion of neighbourhood.

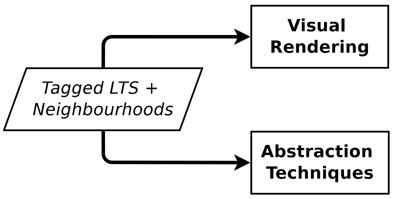


Figure 4.6: Neighbourhood exploitation methods.

Methodology As far as usability is concerned, here is what we advocate for using our approach from a methodological perspective. First, the developer can use the visual rendering for taking a global look at the erroneous part of the tagged LTS and possibly notice interesting structures in that LTS that may guide her to specific kinds of bugs. The developer can dive in the LTS by focusing on chosen states or neighbourhoods and use the step-by-step animation features for that exploration. Moreover, she can add to the visualization some specific counterexample in order to focus on a particular trace of interest (the shortest counterexample for instance) and use the visualization functionalities to better understand the transitions and neighbourhoods on that specific trace.

In a second time, the developer can investigate more in detail the bug behaviour by focusing on single counterexamples. To do this, the developer can use one of the proposed abstraction techniques. Abstraction techniques aim at simplifying the counterexample produced from the LTS and a given property, making a joint analysis of the counterexample and of the tagged LTS with the set of neighbourhoods previously computed.

4.3.1 Visualization Techniques

The visualization techniques we developed give to the developer a graphical representation of the tagged LTS extended with neighbourhoods, where correct/incorrect/neutral transitions and neighbourhoods are highlighted. These 3D visualization techniques make use of different colours to distinguish correct (green), incorrect (red) and neutral (black) transitions on the one hand, and all kinds of neighbourhoods (represented with different shades of yellow) on the other hand. The goal of this visual rendering is to provide a support for visualizing the erroneous part of the tagged LTS and emphasize all the neighbourhoods where a choice is taken and makes the specification either head to correct or incorrect behaviour. Moreover, the developer can possibly notice interesting structures in the LTS that may guide her to recognize specific kinds of bugs. We show several specifications in Chapter 8 with invalid properties, which allow seeing how our approach can be used in practice to visually identify typical bugs.

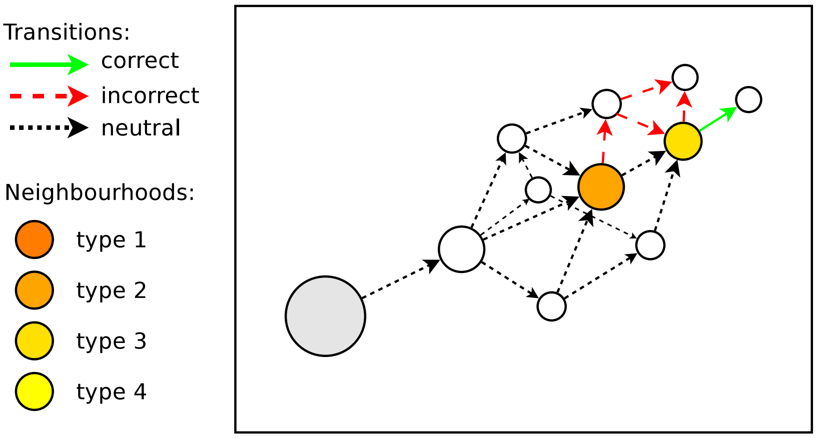


Figure 4.7: 3D visualization.

Figure 4.7 presents a mock-up of our 3D visualization approach. The state on the left hand side of the figure depicted in light grey represents the initial state of the LTS, while the two states in yellow and orange correspond to two neighbourhoods. Transitions are here highlighted according to the types presented in Section 4.1. Different neighbourhoods are also highlighted according to the categorization presented in Section 4.2.1.

Beyond visualizing the whole erroneous LTS, the visualization techniques also provide functionalities in order to explore the tagged LTS for debugging purposes, facilitating the manipulation of such LTS. For instance, the developer can rotate the LTS to change her point of view, or perform zoom in/out on given states and neighbourhoods. We present here the two main functionalities of the tool: the step-by-step path traversal and the counterexample highlighting.

Step-by-Step Path Traversal One of the functionalities presented by the 3D visualization technique provides a path traversal functionality that performs step-by-step animation starting from the initial state or from any chosen state in the LTS. This functionality allows the exploration of paths that lead to given neighbourhoods. The traversal keeps track of the already traversed states and transitions. Moreover, the developer can also move backward in that trace, performing different choices. Figure 4.8 shows a mock-up of the step-by-step path traversal functionality.

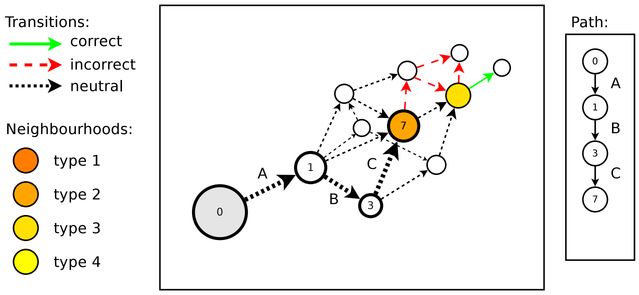


Figure 4.8: Step-by-step path traversal functionality.

Counterexample Highlighting Another functionality allows the developer to visualize a specific counterexample in the tagged LTS. The counterexample (produced by the same tagged LTS under analysis) can first be chosen by the developer, thus is loaded and highlighted in the tagged LTS. Then, this counterexample can be traversed using the step-by-step animation feature for exploring specific details of such trace (in particular, correct/incorrect transitions and neighbourhoods). Figure 4.9 shows a mock-up of the counterexample highlighting functionality.

4.3.2 Abstraction Techniques

We present here some abstraction techniques we have proposed for our approach, organised into two sets. The first set of techniques is independent of the model. It includes: (i) the

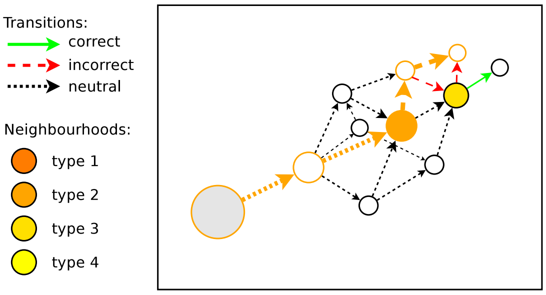


Figure 4.9: Counterexample highlighting functionality.

abstracted counterexample technique, that allows to remove from a counterexample actions that do not belong to neighbourhoods (and thus represent noise); (ii) the *shortest path to a neighbourhood* technique, which retrieves the shortest sequence of actions that leads to a neighbourhood. The second set is dependent on the model, consisting of techniques that need an action or a pattern of actions to perform the analysis. It details improved versions of (i) and (ii), where the user provides a pattern representing a sequence of non-contiguous actions, in order to allow the developer to focus on a specific part of the model. Note that the sets of techniques we present here can be enhanced with additional ones. The abstraction techniques we have developed can be used as basis for new ones, by combining them or by refining their results with constraints on the neighbourhood types. We will comment on the relevance and benefit of the described abstraction techniques on real-world examples in Chapter 8.

Abstraction Techniques Independent of the Model

We present in this section some abstraction techniques that are independent of the model, meaning that they do not require an additional input from the developer.

Abstracted Counterexample We are able to provide an abstraction of a given counterexample by keeping only transitions that belong to neighbourhoods. The aim of this abstraction technique is to enhance the information usually contained in a counterexample by pointing out actions involved in the cause of the bug. Given the tagged LTS M_T, obtained from a model M and a property φ, the set of states $S_N \subset S_T$ where neighbourhoods have been identified, and a counterexample c_e, produced from M and φ, the procedure for the counterexample abstraction consists of the following steps:

1. Matching between states of c_e with states of M_T.

2. Identification of states in c_e that match states in S_N.

3. Suppression of actions in c_e, which do not represent incoming or outgoing transitions of a neighbourhood.

In the case of liveness properties, the counterexample can be in the form of a lasso. In order to treat such counterexample, the lasso is first unrolled. Then the unrolled counterexample is treated as above.

Figure 4.10 shows an example of a counterexample where two neighbourhoods, highlighted on the right side, have been detected and allow us to identify actions that are preserved in the abstracted counterexample. For illustration purposes, let us consider the counterexample, produced by a model checker from a model M and a property φ, given on the top part of Figure 4.10. Once the set of neighbourhoods in the tagged LTS is computed using M and φ, we are able to locate sub-sequences of actions corresponding to transitions in the neighbourhoods. We finally remove all the remaining actions to obtain the abstracted counterexample shown on the bottom part of the figure.

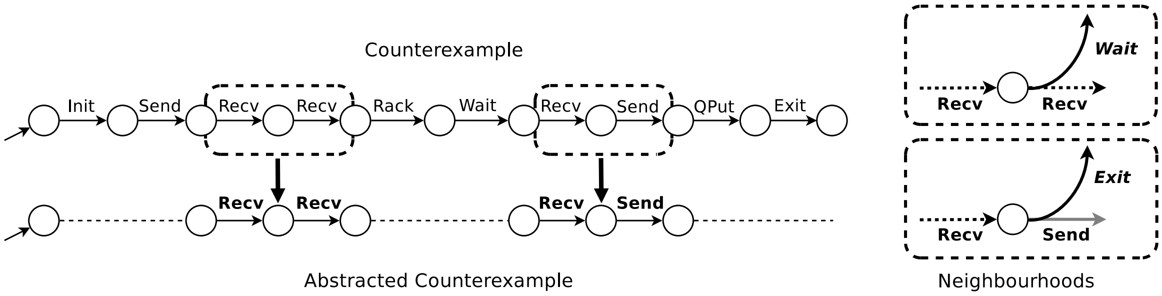


Figure 4.10: Counterexample abstraction.

Some alternative versions can be derived from this abstraction technique. An alternative allows to refine the result of the technique by returning only actions that belong to a given kind of neighbourhood. For instance, we can abstract the counterexample by showing only actions that belong to neighbourhoods with incorrect transitions.

Shortest Path to a Neighbourhood The aim of this abstraction technique is to provide a starting point to debug models that contain a high number of neighbourhoods. The shortest path to a neighbourhood indeed shows the simplest way to reach the first choice that may cause the bug. Given the tagged LTS M_T, obtained from a model M and a property φ, the set of states with a neighbourhood $S_N \subset S_T$, and the set of all traces $t(S_N) \subset t_\delta(M_T)$ between s_T^0 and each $s \in S_N$, we extract the trace of size $n \in \mathbb{N}$ where n is minimal w.r.t. the size of all other traces in $t(S_N)$. The neighbourhood is the closer one to s_T^0 in terms of number of transitions between s_T^0 and the state where the neighbourhood has been identified. From a practical point of view, this consists in performing a breadth-first search for a neighbourhood in the tagged LTS, and then retrieving the sequence of actions that are needed to reach that neighbourhood from the initial state s_0. Let us consider a portion of a tagged LTS M_T depicted in Figure 4.11. States in grey highlight the set of neighbourhoods identified in M_T. We search for the neighbourhood that is the closest one to the initial state. Transitions highlighted in black show the shortest path from the initial state to the closest neighbourhood.

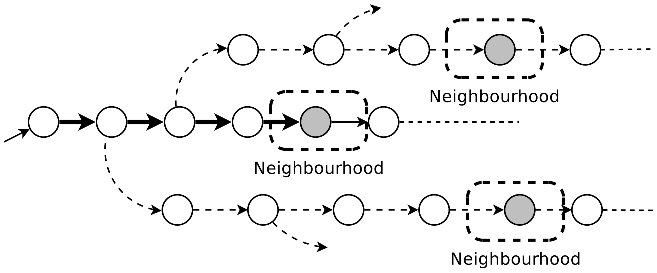


Figure 4.11: Extracting the shortest path between the initial state and a neighbourhood.

Similarly to the abstracted counterexample technique, we define alternative versions, by refining the searched type of neighbourhood or by forcing the path to match a pattern of non-contiguous actions. A refined version of this technique consists of retrieving the shortest path from the initial state to a neighbourhood with incorrect transitions. This alternative version is useful to discover the shortest path that leads to a choice where at least one of the possible options (that is, the one given by the incorrect transition) triggers a behaviour that does not satisfy the given property.

Abstraction Techniques Dependent on the Model

We present here alternative versions of the two previous abstraction techniques in which we exploit an input given by the user, in the form of a pattern representing a sequence of non-contiguous actions. Note that the pattern can also be composed of a single action. The aim of these techniques is to help the developer to focus on a specific part of the analysed system to check whether it is involved in the bug.

Abstracted Counterexample Through a Given Pattern This alternative version of the abstracted counterexample technique exploits an input given by the user, in the form of a pattern representing a sequence of non-contiguous actions, to help the developer to focus on a specific part of the analysed system. To do this, we produce the counterexample that matches the given pattern and then we abstract it. Given the tagged LTS M_T, given a pattern r of size $n \in \mathbb{N}$ in the form of a sequence of non-contiguous labels $l_1, l_2, \ldots, l_n \in \Sigma_T$, provided by the user, we apply the procedure described in the abstracted counterexample technique (see the abstraction techniques independent on the model) to a counterexample c_e that matches the given pattern r to obtain the abstracted counterexample. Note that the pattern can also be composed of a single action.

This technique may help in understanding the relevance of the given pattern of actions in the cause of the bug. Indeed, let us imagine that at least one of the actions that belongs to the pattern is present in a neighbourhood with an incorrect transition. In this case, the pattern of actions is associated to the cause of the bug. Let us now imagine an example where the pattern (for simplicity, composed of a single action) belongs to the counterexample, but it is placed after an incorrect transition, like the one depicted in

Figure 4.12. In this case, the actions in the pattern occur when the execution is already intended to trigger the bug, meaning that it is not relevant from a debugging perspective.

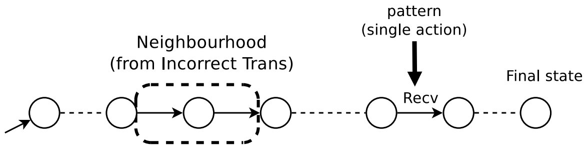


Figure 4.12: Pattern (of one action) in the abstracted counterexample.

Shortest Path to a Neighbourhood Through a Given Pattern This abstraction technique searches for specific paths from the initial node to a neighbourhood that match a given pattern in the form of a sequence of non-contiguous actions, representing a variation of the shortest path to a neighbourhood described in Section 4.3.2. Given the tagged LTS M_T, obtained from a model M and a property φ, the set of states with a neighbourhood $S_N \subset S_T$, and the set of all traces $t(S_N) \subset t_\delta(M_T)$ between s_T^0 and each $s \in S_N$, given a pattern r of size $n \in \mathbb{N}$ in the form of a sequence of non-contiguous labels $l_1, l_2, \ldots, l_n \in \Sigma_C$, provided by the user, we define as $t(r) \subset t(S_N)$ the subset of traces between s_C^0 and each $s \in S_N$ that matches the given pattern r. We then extract the trace of size $m \in \mathbb{N}$ where m is minimal w.r.t. the size of all other traces in $t(r)$.

For illustration purposes, let us consider again the portion of a tagged LTS M_T depicted in Figure 4.13. States in grey highlight the set of neighbourhoods identified in M_T. We search for the path between the initial state and one of the neighbourhoods, that matches a sequence of actions *Recv* and *Rack* with possibly other actions in between. Transitions highlighted in black show the path that matches the given pattern.

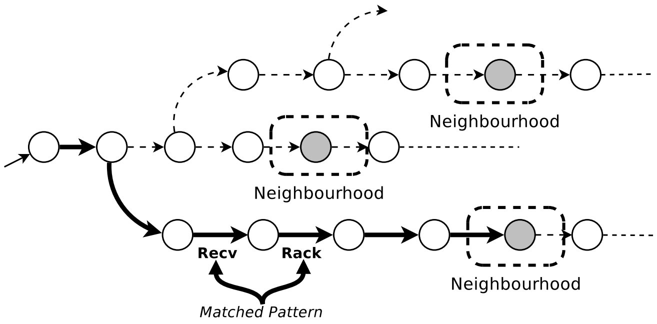


Figure 4.13: Extracting the shortest path to a neighbourhood that matches a given pattern.

This technique can be helpful to check if the shortest path that leads to a neighbourhood contains some actions provided by the user. Note that if the searched pattern of actions is placed after a neighbourhood, the technique will not return any result.

Chapter 5

The Counterexample LTS Approach

In this chapter we describe the approach for computing the tagged LTS for safety property violations. We first introduce the procedure to build an LTS containing all counterexamples (*counterexample LTS*), given a model of the system (which we call *full LTS*, in order to distinguish it from the counterexample LTS) and a safety property. We then present a method to match states of the counterexample LTS and states of the full LTS. This matching information allows us to identify transitions at the *frontier* between the counterexample and the full LTS. The frontier is the area where traces, that share a common prefix in the two LTSs, split in different paths. The computation of this frontier first allows to detect correct transitions, which are then added to the counterexample LTS, making it an enriched counterexample LTS. In a second step the enriched counterexample LTS is used to detect incorrect and neutral transitions, leading to the construction of the tagged LTS.

The main steps of the Counterexample LTS approach are summarised in Figure 5.1.

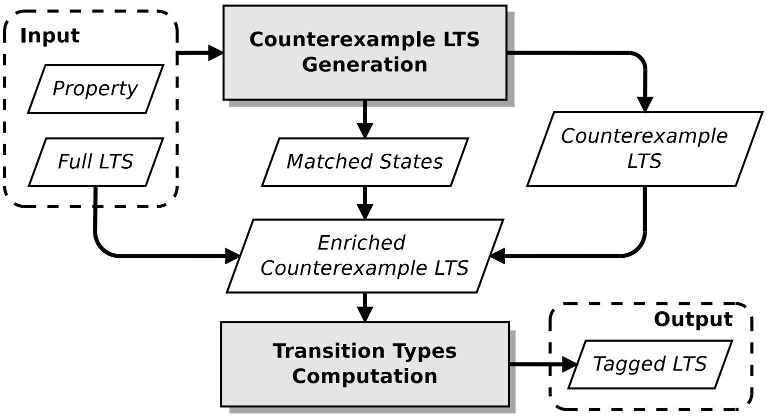

Input
Property
Full LTS
Counterexample LTS Generation
Matched States
Counterexample LTS
Enriched Counterexample LTS
Transition Types Computation
Output
Tagged LTS

Figure 5.1: Counterexample LTS approach overview.

At the end of the chapter we show some examples of neighbourhoods extracted from the tagged LTS produced with the Counterexample LTS approach.

5.1 Counterexample LTS Generation

The full LTS (M_F) is given as input in our approach and is a model representing all possible executions of a system. Given such an LTS and a safety property, our goal in this subsection is to generate the LTS containing all counterexamples (M_C).

Definition 14 *(Counterexample LTS) Given a full LTS $M_F = (S_F, s_F^0, \Sigma_F, T_F)$, where $\delta \notin \Sigma_F$, and a safety property φ, a counterexample LTS M_C is an LTS such that $t_\delta(M_C) = t(M_F) \cap t_\varphi$, i.e., a counterexample LTS is a finite representation of the set of all traces of the full LTS that violate the property φ.*

We use the set of final traces $t_\delta(M_C)$ instead of $t(M_C)$ since $t(M_C)$ is prefix closed, but prefixes of counterexamples that belongs to $t(M_C)$ are not counterexamples. Moreover, traces in the counterexample LTS share prefixes with correct traces in the full LTS.

Let us illustrate the idea of counterexample LTS on the example given in Figure 5.2. The full LTS on the left hand side represents a model of a simple protocol that performs *Send* and *Receive* actions in a loop. The counterexample LTS on the right hand side is generated with a property φ stating that *"no more than one Send action is allowed"*. Note that final transitions characterised by the δ label are not made explicit in this example.

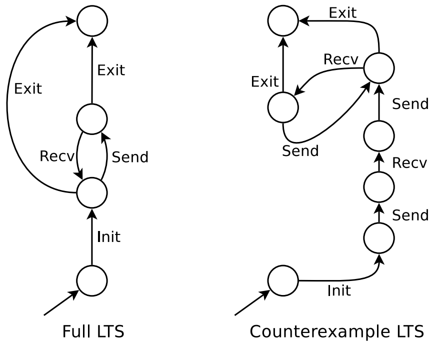


Figure 5.2: Full LTS and counterexample LTS.

Given a full LTS M_F and a safety property φ, the procedure for the generation of the counterexample LTS consists of the following steps:

Step a) Conversion of the φ formula describing the property into an LTS called M_φ, using the technique that allows the encoding of a formula into a graph described in [LM13]. Given an action formula that represents a logical formula built from basic action predicates and Boolean operators, this technique builds the LTS by replacing action formulas with finite sets of transitions that can potentially occur in the process composition. M_φ is a finite representation of t_φ, using final transitions, such that $t_\delta(M_\varphi) = t_\varphi \cap \Sigma_F^*$, where Σ_F is the set of labels occurring in M_F. In this step, we also determinise M_φ, as defined in Section 3.4.2, and we finally reduce the size of M_φ without changing its behaviour, performing a minimization based on strong bisimulation [Mil89]. Those two transformations keep the set of final traces of M_φ unchanged. The LTS M_φ obtained in this way is the minimal one that is deterministic and accepts all the execution sequences that violates φ.

Let us consider again the previous example. The property φ that states that no more than one *Send* action is allowed is translated in the LTS depicted in Figure 5.3a. The asterisk symbol $*$ is used here for simplicity to summarise all the transitions that are represented by the labels which are not contained in the property, while the δ symbol points out the final transitions.

Step b) Synchronous product between M_F and M_φ with synchronisation on all the labels of Σ_F (thus excluding the final label δ). The result of this product is an LTS whose final traces belong to $t(M_F) \cap t_\delta(M_\varphi)$, thus it contains all the traces of the LTS M_F that violate the formula φ. Note that $t(M_F) \cap t_\delta(M_\varphi) = t(M_F) \cap t_\varphi$, because $t(M_F) \subseteq \Sigma_F^*$ and $t_\delta(M_\varphi) = t_\varphi \cap \Sigma_F^*$. Figure 5.3b shows the result of the product of the full LTS depicted in Figure 5.2 and the property φ in the form of an LTS depicted in Figure 5.3a.

Step c) Pruning of the useless transitions generated during the previous step. In particular, we use the pruning algorithm proposed in [MPS12] to remove the traces produced by the synchronous product that are not the prefix of any final trace. As we can see in Figure 5.3b, final traces end with the δ transitions (that have been introduced by the final δ transition in M_φ produced in the first step). We first remove all the transitions that do not belong to final traces. In the example, these consist of the *Exit* transition after the *Init* transition and the *Exit* transition after the first *Send* and *Recv* transitions. At the end of this process the δ transitions are not needed any more, thus they are removed. The result of this step is depicted in Figure 5.3c.The LTS M_C obtained at the end of *Step c* is thus a counterexample LTS for M_F and φ.

5.2 States Matching

We now need to match each state belonging to the counterexample LTS with the corresponding one in the full LTS. To do this, we define a matching relation between states of the two LTSs, by relying on the simulation relation introduced in Chapter 3. In our context, we want to build such a relation between M_C and M_F, where a state $x \in S_C$ matches a state $y \in S_F$ when the first is simulated by the latter, that is, when $x \sqsubseteq y$.

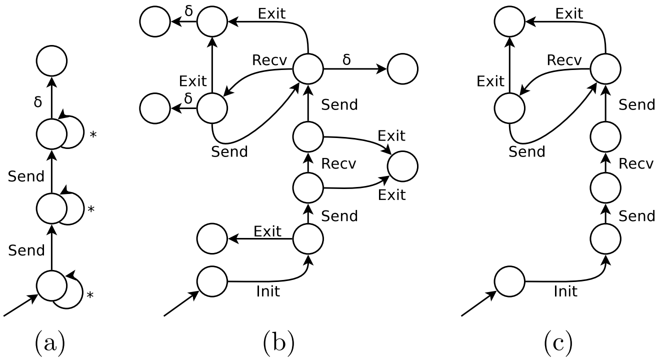


Figure 5.3: The Counterexample LTS generation steps.

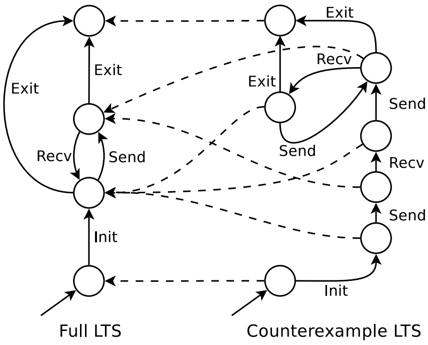


Figure 5.4: States matching.

Since the LTS that contains the incorrect behaviours is extracted from the full LTS, the full LTS always simulates the counterexample LTS. Note that the correspondence of states between the counterexample LTS and the full LTS is *n-to-1*. Indeed multiple states of the counterexample LTS may correspond to a single state of the full LTS. For instance this is the case when a loop is partially rolled out.

To build the simulation relation between M_C and M_F we exploit information generated by the synchronous product used in step b) of Section 5.1. Indeed, the product also generates a list of couples of states of M_F and M_φ where each couple is associated to the resulting state of M_C, representing the matching between each state of M_C with the corresponding one in M_F. Let us consider again the example described in Figure 5.2. Each state of the counterexample LTS on the right hand side of the picture matches a state of the full LTS on the left hand side as shown in Figure 5.4. Note that the property φ has become unsatisfied after a given number of iterations of the loop composed of *Send* and *Recv* actions, resulting in a correspondence of several states of the counterexample LTS to a single state of the full LTS.

5.3 Transition Types Computation

The state matching information, retrieved in Section 5.2, is here exploited as input to compare transitions outgoing from similar states in both LTSs. This comparison aims at identifying transitions that originate from matched states, and that appear in the full LTS but not in the counterexample LTS. We call this kind of transition a *correct transition.*

Definition 15 *(Correct Transition) Given an LTS $M_F = (S_F, s_F^0, \Sigma_F, T_F)$, a property φ, the counterexample LTS $M_C = (S_C, s_C^0, \Sigma_C, T_C)$ obtained from M_F and φ, and given two states $s \in S_F$ and $s' \in S_C$, such that $s' \sqsubseteq s$, we call a transition $s \xrightarrow{l} s'' \in T_F$ a correct transition if there is no transition $s' \xrightarrow{l} s''' \in T_C$ such that $s''' \sqsubseteq s''$.*

A correct transition is preceded by incoming transitions that are common to the correct and incorrect behaviours, meaning that they appear in both the full and the counterexample LTSs. These transitions are *relevant predecessors*, which we already presented in Section 4.2. States where a correct transition exists allow us in a second step to identify neighbourhoods belonging to the first type (according to the neighbourhood taxonomy of Chapter 4).

Let us illustrate the detection of a correct transition on an example. Figure 5.5 shows a piece of full LTS and the corresponding counterexample LTS. The full LTS on the left hand side of the figure represents a state that has been matched by a state of the counterexample LTS on the right hand side and it has correct transitions outgoing from it.

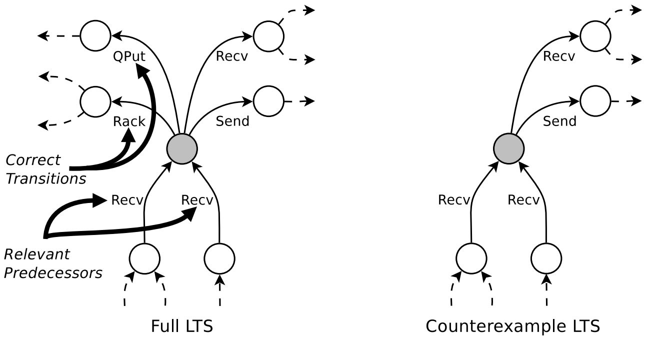


Figure 5.5: Example of correct transitions.

We then add all the correct transitions detected in the full LTS to the counterexample LTS. Note that correct transitions added to the counterexample LTS are all directed to a new dedicated *sink state* (s_k). In this way the behaviour described by the counterexample LTS is not altered. The counterexample LTS, with the added correct transitions, is called *enriched counterexample LTS.*

Definition 16 *(Enriched Counterexample LTS) Given the counterexample LTS $M_C = (S_C, s_C^0, \Sigma_C, T_C)$ obtained from a full LTS M_F and a property φ, the set of correct transitions T_{ct} detected in M_F and the set of labels Σ_{ct} in T_{ct}, the enriched counterexample LTS is a* *tuple* $M_{EC} = (S_{EC}, s^0_{EC}, \Sigma_{EC}, T_{EC})$ *where* $S_{EC} = S_C \cup s_k$*,* $s^0_{EC} = s^0_C$*,* $\Sigma_{EC} = \Sigma_C \cup \Sigma_{ct}$*, and* $T_{EC} = T_C \cup T_{ct}$*.*

In Figure 5.6 we illustrate a portion of enriched counterexample LTS. The two correct transitions detected in the full LTS are added to the counterexample LTS and are directed to a destination state represented by the sink state.

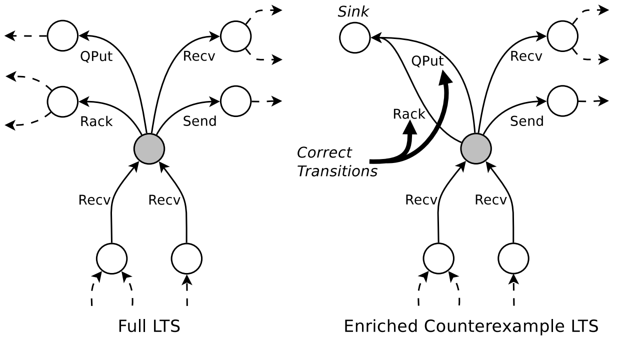


Figure 5.6: Example of portion of Enriched Counterexample LTS.

All the information we need to perform the following steps is thus contained in the sole enriched counterexample LTS. We now focus on transitions that lead only to behaviours that do not satisfy the property. To detect this kind of transitions we check each transition in the enriched counterexample LTS searching for correct transitions among its subsequent transitions. If this is not the case, we classify the transition as *incorrect transition.*

Definition 17 *(Incorrect Transition) Given an enriched counterexample LTS* $M_{EC} = (S_{EC}, s^0_{EC}, \Sigma_{EC}, T_{EC})$*, a state* $s \in S_{EC}$*, the set* S_{succ} *that contains* s *and its successors states, we call a transition* $t = s \xrightarrow{l} s' \in T_C$ *an incorrect transition if there is no state* $s' \in S_{succ}$ *such that* $\exists t' = s' \xrightarrow{l} s'' \in T_C$ *and* t' *is a correct transition.*

Definition 17 produces sequences of incorrect transitions, since successors of incorrect transitions are also incorrect. Note that this is not the case for correct transitions. Since correct transitions are all directed to the sink state, they do not have successors and consequently they do not produce any sequences of actions.

The transitions that are not correct nor incorrect are the ones that have both correct and incorrect transitions among their successors. We call these transitions *neutral transitions.* The detection of the three types of transition on the enriched counterexample LTS allows us to obtain a tagged LTS (introduced in Section 4.1). Figure 5.7 shows the portion of tagged LTS computed from the portion of enriched LTS of Figure 5.6.

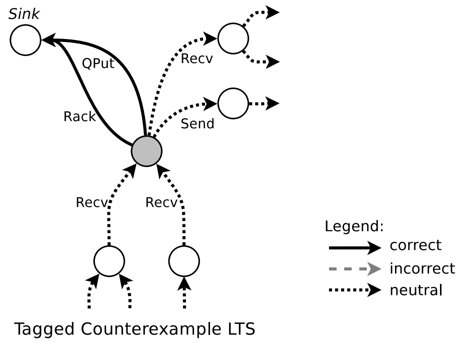


Figure 5.7: Example of portion of tagged Counterexample LTS.

5.4 Neighbourhood Examples

The tagged LTS allows us to extract neighbourhoods in a second step, according to the procedure detailed in Section 4.2. The neighbourhood computation procedure detect neighbourhoods in states where an incoming neutral transition is followed by a correct or an incorrect one. Let us show a couple of neighbourhoods examples.

We first take into account again the last example of portion of tagged LTS. The incoming and outgoing transitions of the state in the centre of Figure 5.8 represent a neighbourhood, which belongs to the first type (see the taxonomy in Section 4.2.1).

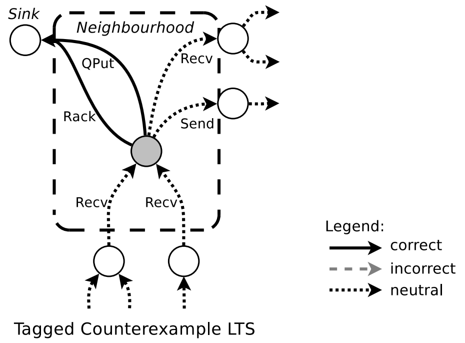


Figure 5.8: Example of neighbourhood in a tagged Counterexample LTS.

Let us now consider again the counterexample LTS generated in Figure 5.3. We add the sink state with correct transitions to such counterexample LTS, and we detect all the other transitions types. In Figure 5.9 we depict the resulting tagged LTS, where we identify in a second step some neighbourhoods. Correct transitions, all representing the *Exit* transition, highlight a neighbourhood in the corresponding source state. In this example, the first two neighbourhoods belong to the first type of neighbourhood (according to the taxonomy detailed in Section 4.2.1). The third neighbourhood has both a correct and an incorrect transition, corresponding to the second *Send* transition. Consequently, this neighbourhood belongs to type 3. Note that this example contains an high number of neighbourhoods w.r.t. the number of states in the LTS. This is commonly not the case in real-world cases, since other transitions considered as noise by the developer are usually present between neighbourhoods in tagged LTSs.

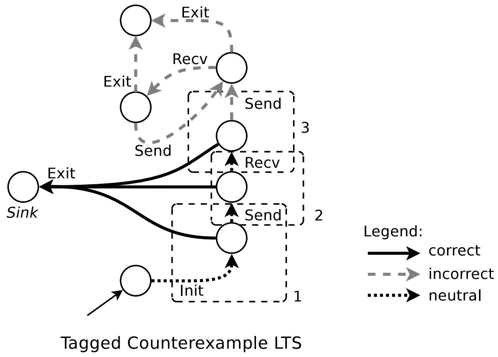


Figure 5.9: Neighbourhood identification.

Chapter 6

The Prefix-Suffix Approach

In this chapter we describe our approach to handle liveness properties, in particular inevitability properties, which are treated as a sequence of inevitable actions. The Counterexample LTS technique presented in Chapter 5 does not work with liveness properties violations. The reason is that the procedure which generates the counterexample LTS does not support the presence of lassos in the counterexamples produced from a liveness property violation. We thus propose an alternative approach, called the Prefix / Suffix approach, which focuses on the analysis of the execution of the property's actions in each LTS state to locate neighbourhoods.

We first present in Section 6.1 the notions of prefixes and suffixes. These notions are used to capture information about the actions belonging to the property that have already been or remain to be executed. In Section 6.2 we describe the algorithms we developed to compute prefixes and suffixes by analysing SCCs in the original LTS, in order to obtain an augmented version of such LTS (*augmented LTS*). Prefix and suffix information is successively used to detect the type of the transition (correct, incorrect or neutral) in order to obtain the tagged LTS introduced in Chapter 4. The main steps of the Prefix / Suffix approach are summarised in Figure 6.1. At the end of the chapter we illustrate with an example of neighbourhood detected in a tagged LTS computed with the Prefix / Suffix approach, and we compare this approach to the Counterexample LTS one.

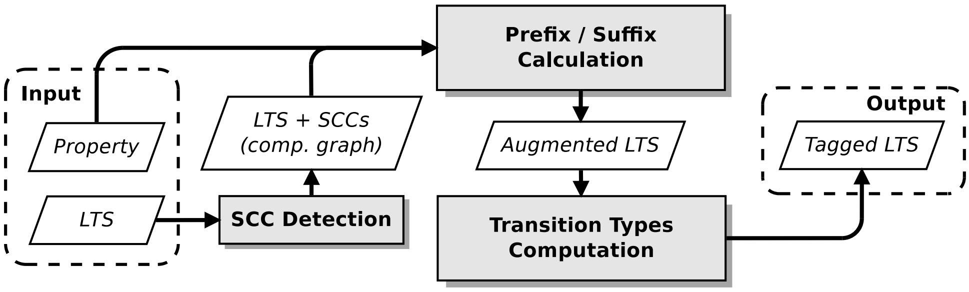


Figure 6.1: Prefix / Suffix approach overview.

6.1 Prefixes and Suffixes

An LTS M is a model representing all possible executions of a system. Given an inevitable execution property φ, our goal is to analyse each state s in M to understand whether all the traces that pass through s satisfy the sequence of actions expressed by φ. To do this we compare the prefixes of traces that reach the state to prefixes of the given sequence. Similarly we compare the suffixes of traces that start from the state to suffixes of the given sequence. Note that in this work we use the symbol " $\cdot$ " to denote the concatenation operator for labels and sequences of labels.

Definition 18 *(Sequence of Inevitable Actions) Given an inevitable execution property $p = \texttt{Inev}(l_1,\ \ldots,\ l_n)$, the sequence of concatenated labels $k = l_1 \cdot l_2 \cdot \ldots \cdot l_n$ of size $n \in \mathbb{N}$ is the sequence of inevitable actions that respect the order defined by the nested inevitability operator.*

The sequence of inevitable actions may represent non-contiguous transitions in the model. In order to match traces and prefixes (suffixes, resp.) of traces with the sequence of inevitable actions, we define a matching operator as follows:

Definition 19 *(Matching Operator) Given an LTS $M = (S, s^0, \Sigma, T)$, a sequence of labels $j = a_1 \cdot a_2 \cdot \ldots \cdot a_n$, a sequence of contiguous transitions $z = s_1 \xrightarrow{l_1} s_2 \in T, s_2 \xrightarrow{l_2} s_3 \in\ T, \ldots, s_{m-1} \xrightarrow{l_{m-1}} s_m \in T$, z is said to match j, written $j \prec z$, if there exists integers $1 \leq i_1 < i_2 < \ldots < i_n \leq m$ such that $a_1 = l_{i_1},\ a_2 = l_{i_2},\ \ldots,\ a_n = l_{i_n}$.*

We assign to each state of the LTS the prefixes of the sequence of inevitable actions k obtained up to the state under analysis. To do this, we introduce the notions of *max prefix* and *common prefix*, w.r.t. k. The max prefix is the longest prefix of the k sequence among the prefixes of traces that end in a given state. The common prefix is the longest prefix of the k sequence that is common to all the prefixes of traces that end in a given state. We define $\mathbb{T}^e_s$ as the set of all the prefixes of traces that end in s and $\mathbb{P}^k$ as the set of all the prefixes of k.

Definition 20 *(Max and Common Prefix) Given an LTS $M = (S, s^0, \Sigma, T)$, a sequence of inevitable actions k, the set $\mathbb{P}^k$ of all the prefixes of k, a state $s \in S$, the max prefix, defined as mp_s, is the longest element in $\mathbb{P}^k$ such that $\exists t \in \mathbb{T}^e_s,\ mp_s \prec t$. The common prefix, defined as cp_s, is the longest element in $\mathbb{P}^k$ such that $\forall t \in \mathbb{T}^e_s,\ cp_s \prec t$.*

In a similar way we assign to each state of the LTS the suffixes of the sequence of inevitable actions k that will be completed starting from s. We introduce the notions of *max suffix* and *common suffix*, w.r.t. k. The max suffix is the longest suffix of the k sequence among the suffixes of traces that start from a given state. The common suffix is the longest suffix of the k sequence that is common to all the suffixes of traces that start from a given state. We define $\mathbb{T}^o_s$ as the set of all the suffixes of traces that start from s and $\mathbb{S}^k$ as the set of all the suffixes of k.

Definition 21 *(Max and Common Suffix) Given an LTS $M = (S, s^0, \Sigma, T)$, a sequence of inevitable actions k, the set $\mathbb{S}^k$ of all the suffixes of k, a state $s \in S$, the max suffix, defined as ms_s, is the longest element in $\mathbb{S}^k$ such that $\exists t \in \mathbb{T}^o_s,\ ms_s \prec t$. The common suffix, defined as cs_s, is the longest element in $\mathbb{S}^k$ such that $\forall t \in \mathbb{T}^o_s,\ cs_s \prec t$.*

The example given in Figure 6.2 shows the max/common prefixes and suffixes calculated on each state of an LTS for a given sequence of inevitable actions $k = A \cdot Y$. Let us take a look at state 8: the *cp* value shows that the action A exists in every prefix of k produced by prefixes of traces that end in state 8. Conversely, the *cs* value in state 8 is empty while the *ms* value is $A \cdot Y$, meaning that the suffix $A \cdot Y$ is not contained in every suffix of traces that starts in state 8. It is also interesting to notice that in state 3 all the mp, cp, ms, cs values are empty, since none of the actions in k have been or will eventually be met. As a matter of fact, we can see that the only suffix of traces that respects the k sequence is the one that begins with the transition $9 \xrightarrow{C} 10 \in T$, since the composition of *cp* in state 9 and *cs* in state 10 represents the whole k sequence (we will see in details how the max / min prefixes and suffixes are used to categorise a given transition in Section 6.3). Note that the repetition of the action A in some traces does not increase the size of computed prefixes, since they must represent a prefix of the given k sequence.

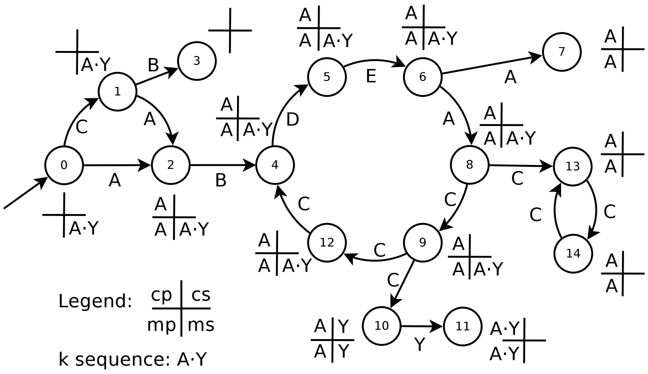


Figure 6.2: Max / common prefixes and suffixes.

In some cases inevitable execution properties might not be satisfied because of loops in which the execution of the system can remain infinitely. Our notion of suffix allows us to discover such loops and understand whether they prevent the satisfaction of the property. In particular, given a state s, if a loop impedes the completion of the k sequence in suffixes of traces starting in s, the *ms* value is empty. If a choice that allows to exit from the loop and later complete the k sequence exists inside the same loop, the *ms* value consists of the suffix needed to complete the k sequence from s. Note that in both cases the *cs* value in s is empty. One of these loops is present in the example in Figure 6.2 and is composed of states 4, 5, 6, 8, 9 and 12. These loops are treated in the next section by extracting the *Strongly Connected Components (SCCs)* [Tar72] from the LTS. The LTS with the max/common prefixes (suffixes, resp.) computed for each state is called *augmented LTS.*

Definition 22 *(Augmented LTS) Given an LTS $M = (S, s^0, \Sigma, T)$ and a sequence of in-*

evitable actions k, the augmented LTS is a tuple $M^k_E = (S_E, s^0_E, \Sigma_E, T_E)$ such that each state $s_E \in S_E$ is a tuple $s_E = (s, mp_s, cp_s, ms_s, cs_s)$, where $s \in S$, $mp_s, cp_s \in \mathbb{P}^k_s$, $ms_s, cs_s \in \mathbb{S}^k_s$; $s^0_E = (s^0, mp_{s^0}, cp_{s^0}, ms_{s^0}, cs_{s^0})$; $\Sigma_E = \Sigma$; $T_E \subseteq S_E \times \Sigma_E \times S_E$, where $\forall s \xrightarrow{l} s' \in T$, $(s, mp_s, cp_s, ms_s, cs_s) \xrightarrow{l} (s', mp_{s'}, cp_{s'}, ms_{s'}, cs_{s'}) \in T_E$.

6.2 Prefixes and Suffixes Calculation

This section presents the computation of the prefixes and suffixes defined in Section 6.1. In order to handle cycles in an LTS we use the notion of SCC. To detect all the SCCs in an LTS we use the Tarjan's SCCs algorithm [Tar72], detailed in Chapter 3. Given a sequence of inevitable actions k, our approach considers each SCC of the LTS, and computes the max/common prefixes and suffixes for every state in the SCC. Note that we start computing prefixes for states in a given SCC only when all its predecessor SCCs have been computed (in the case of suffixes we first compute all the successors).

We now introduce some definitions that we will use throughout the whole section. Given an LTS $M = (S, s^0, \Sigma, T)$ and an SCC $G = (S_G, \Sigma_G, T_G)$ in M, we denote as $S^e_G \subseteq S_G$ the set of *initial states* of G, such that, given a transition $s \xrightarrow{l} s' \in T$, the state $s \notin S_G$ and $s' \in S^e_G$. The transition $s \xrightarrow{l} s' \in T$ is defined as *incoming transition* and the set of incoming transitions is written as T^e_G. Similarly, we denote as $S^o_G \subseteq S_G$ the set of *outgoing states* such that, given a transition $s \xrightarrow{l} s' \in T$, the state $s \in S^o_G$ and $s' \notin S_G$. The transition $s \xrightarrow{l} s' \in T$ is defined as an *outgoing transition* and the set of outgoing transitions is written as T^o_G.

Moreover, we denote as $\mathbb{G}_\mathbb{M}$ the *component graph* [CJLV02] of an LTS M where the states are given by SCCs of M. The SCC containing the initial state s_0 of the LTS M does not have any predecessors and it is defined as G^0. By definition, since all cycles are contained in SCCs, $\mathbb{G}_\mathbb{M}$ is a directed acyclic graph. The rest of this section presents the computation of the max prefix (suffix, resp.) and of the common prefix (suffix, resp.) for states of an SCC.

6.2.1 Max Prefix Calculation

The max prefix inside an SCC is computed by first extracting the longest max prefix among the incoming states of the SCC. Second, the incoming max prefix is extended with actions contained inside the SCC to produce the longest (possible) prefix of k. Note that the max prefix is the same for all the states of an SCC. The cost of the computation for an SCC G is $O(|T^e_G| + |T_G| + |k|)$, since we first have to explore all the incoming transitions to compute the initial max prefix, and second we have to collect all the actions in the SCC that are also present in k.

Let us consider the SCC composed of states 1, 2 and 3 in Figure 6.3 (note that SCCs in states 0, 4 and 5 are trivial). Given $k = A \cdot B \cdot C$, the initial max prefix for the SCC is A, since the transition from state 0 to state 1 is the only incoming transition and it contains the first action of the k sequence. One can notice that by looping inside the SCC it is possible to complete the k sequence, since the SCC contains also actions B and C. Consequently, the max prefix in each state of the SCC (states 1, 2 and 3) is equivalent to the k sequence.

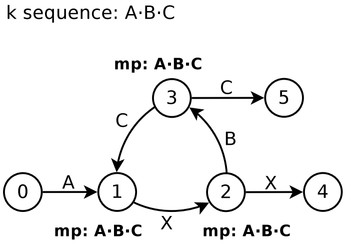


Figure 6.3: Max prefix calculation on an SCC.

6.2.2 Max Suffix Calculation

The max suffix is computed similarly to the max prefix, by considering suffixes of successors instead of prefixes of predecessors. In the example in Figure 6.4 the max suffix for every state in the SCC is $B \cdot C$, since they are the only two actions contained in the SCC that also exist in k.

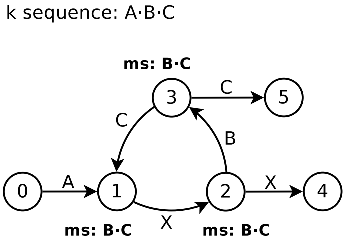


Figure 6.4: Max suffix calculation on an SCC.

6.2.3 Common Prefix Calculation

We describe here the computation of the common prefix for each state in an SCC. The pseudo-code of this procedure is detailed in Algorithm 2. The algorithm is divided into two main steps: initialisation and internal transitions computation.

Initialisation step. Given an SCC G the algorithm initialises the common prefix of states in G to k (Line 3). There are two exceptions to this rule. First, the initial state of the LTS

s^0 is initialised to the empty sequence since it has no predecessors. Second, if s is an initial state of G, cp_s is initialised with the common prefixes of its incoming transitions (Line 7).

Let us take a look at the example in Figure 6.5. The initialisation step assigns $cp = A \cdot B \cdot C$ to states 2 and 3, while it assigns $cp = A$ to state 1, which is the only initial state of the SCC.

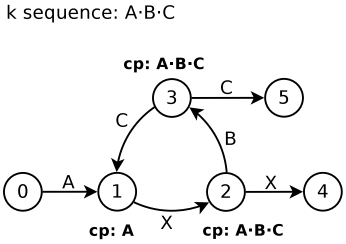


Figure 6.5: Common prefix calculation on an SCC (init step).

Internal transitions computation step. After the initialisation step, all initial states have a common prefix that accounts for transitions from outside of G. However, there may still be paths within G that can produce a smaller prefix. This step deals with internal transitions to detect smaller prefixes. First of all, we use Q (Line 8) as sorted set to order the states in S_G by increasing common prefix size. When modifying the common prefix of a state s, $\textsc{UpdatePosition}(Q, s)$ updates the position of s in Q. The loop of Line 9 iterates on Q, removing the first element s at each iteration. The common prefix of all successors s' of s within G is updated using a function LCP which computes the longest common prefix between two sequences of actions. When producing a smaller prefix, the position of s' in Q is updated.

Let us consider the SCC in Figure 6.6 (initialised in Figure 6.5). The internal transitions computation step corrects the values of cp in states 2 and 3, assigning respectively $cp = A$ and $cp = A \cdot B$. The value of cp in state 1 remains the same, since it was already the lower one among all the states of the SCC.

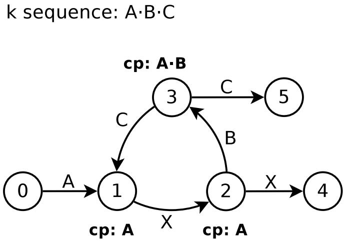


Figure 6.6: Common prefix calculation on an SCC.

Correctness and complexity. The cost of the initialisation step of the algorithm is $O(|S_G| + |T_G^e|)$ since it considers all states and their incoming transitions. The rest of the common prefix algorithm behaves similarly to the Dijkstra's algorithm that deals with the single-source shortest-paths problem in weighted directed graphs (in particular, to the implementation which uses a Fibonacci heap as priority queue). When a state is removed from Q, its common prefix is correct and will no longer be updated, as there is no state in Q with a smaller common prefix. Note that the common prefix of a state s can only be updated twice in the loop. Indeed, the predecessors of s are considered for updates by increasing size of common prefix. There exists either a predecessor of s that has the same common prefix and a transition that does not match in k, or a predecessor that has a common prefix containing one less element, but a transition to s that matches k and increases the prefix. In the first case, it is possible to first encounter a predecessor that generates a smaller common prefix with a matching transition, leading to a first update of cp_s, before the final assignment when reaching the optimal predecessor. The complexity of the loop of the internal transitions computation step is $O(|T_G| + |S_G| \log |S_G|)$, because the while loop performs exactly $|S_G|$ iterations, given that Q is initialised to S_G and an element is removed each time. Moreover the cost of inserting an element to Q and updating its position is $O(\log |Q|)$. Hence, taking into account the initialization and the internal transition computation steps, the complexity of the common prefix algorithm is $O(|T^e_G| + |T_G| + |S_G| \log |S_G|)$

Algorithm 2 Common Prefix Computation

```
procedure CommonPrefix($(G, k)$)
    for all $s \in S_G$ do
        $cp_s \leftarrow k$
        if $s = s^0$ then $cp_s \leftarrow \emptyset$
        else if $s \in S^e_G$ then
            for all $s' \xrightarrow{l} s \in$ ExtractIncomingTrans($(s)$) do
                if $s' \notin S_G$ then $cp_s \leftarrow \mathrm{LCP}(cp_s, cp_{s'} \cdot l)$
    $Q \leftarrow S_G$
    while $Q \neq \emptyset$ do
        $s \leftarrow$ PopFirst($(Q)$)
        for all $s \xrightarrow{l} s' \in T_G$ do
            $t \leftarrow \mathrm{LCP}(cp_s \cdot l, cp_{s'})$
            if $|t| < |cp_{s'}|$ then $cp_{s'} \leftarrow t$ ; UpdatePosition($(Q, s')$)
```

6.2.4 Common Suffix Calculation

The common suffix calculation is similar to the prefix case, but it differs in the initialisation step. In the prefix case, the fact that the execution reaches a state indicates that it was not stuck into a loop of a preceding SCC. Simply transposing this to the prefix case would not work, as it is not certain that the execution will reach a final state. Indeed, in the suffix case the execution may loop into the current SCC and never go through an outgoing transition, and a state may thus have a smaller common suffix than all states from its successor SCCs. This initialisation step is presented in Algorithm 3. In the case of a final state, the suffix is empty (Line 3). Otherwise, the common suffix is initialised to the smallest suffix of k traversed by a loop from s to itself (Line 4). In the absence of loops (SCC with single state and no self-loop), MinSuffixLoop returns k. The remainder of the computation is similar to Algorithm 2, using a function LCS, which computes the longest common suffix, instead of LCP. Searching the smallest-suffix loop for each state is done by iteratively removing labels from k and looking for isolated vertices. Hence, the overall cost of the computation is $O(|T_G^o| + |k| \times (|T_G| + |S_G|) + |S_G| \log |S_G|)$, making the common suffix computation having an higher complexity that the common prefix one.

Algorithm 3 Common Suffix Computation (Initialisation Step)

```
1: procedure CommonSuffixInit($G, k$)
2:     for all $s \in S_G$ do
3:         if $\nexists s \xrightarrow{l} s'$ then $cs_s \leftarrow \emptyset$
4:         else $cs_s \leftarrow$ MinSuffixLoop($s, k, G$)
5:             if $s \in S_G^o$ then
6:                 for all $s \xrightarrow{l} s' \in$ExtractOutgoingTrans($s$) do
7:                     if $s' \notin S_G$ then $cs_s \leftarrow \mathrm{LCS}(cs_s, l \cdot cs_{s'})$
```

Let us consider the example in Figure 6.7. The initialisation step assigns $cs = B \cdot C$ to state 1, which is the smallest suffix of k that can be produced inside the SCC starting from state 1. It then assigns the empty sequence and $cs = C$ to states 2 and 3, since they are outgoing states. The algorithm will later update the value of cs in state 1 to the empty sequence with the internal transitions computation step.

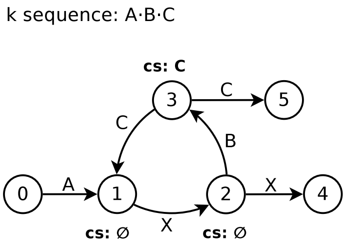


Figure 6.7: Common suffix calculation on an SCC.

6.2.5 Order of Calculation

So far, we have considered the computation of prefixes and suffixes for the states of an SCC. However, evaluating prefixes requires that the prefixes of all predecessor states of an SCC are correct (successors states in case of suffixes). It is thus important to execute our approach on SCCs in an appropriate order. Since by definition there are no cycles in $\mathbb{G}_{\mathbb{M}}$, we can define the *depth* of an SCC as 0 for the SCC that contains s_0, and 1 plus the maximum depth of predecessor SCCs otherwise. Our approach computes prefixes in SCCs by increasing depth, and suffixes by decreasing depth, ensuring the presence of the necessary information. Note that the use of the *depth* value, that guarantees the evaluation of the SCCs in the correct order, makes the calculation of prefixes and suffixes in the initial LTS trivially correct. Given the costs of computing prefixes and suffixes in each SCC, the total cost of the calculation of the augmented LTS is $O(|k| \times (|T| + |S|) + \sum_{G \in \mathbb{G}_{\mathbb{M}}}(|S_G| \log |S_G|))$.

6.3 Transitions Types Computation

We now need to obtain a tagged LTS from the augmented LTS, in order to detect neighbourhoods in a next step. To do this, the augmented LTS with max/common prefixes and suffixes is used to characterise its transitions. A transition is typed as *correct* if it belongs to a correct part of the model, as *incorrect* if it belongs to an incorrect part of the model, as *neutral* if none of the previous cases apply.

More specifically, a *correct transition* leads to a portion of the LTS where the sequence of actions k is always respected. To state whether a transition is a correct one we compute the sum of the length of *cp* in the source state and of *cs* in its destination state. Note that we also have to take into account the label of the transition in this sum, since the concatenation of *cp* with the label may produce a valid prefix of k. If the sum is equal or higher than the size of the k sequence the transition is identified as correct.

Definition 23 *(Correct Transition) Given an augmented LTS* $M_E^k = (S_E, s_E^0, \Sigma_E, T_E)$, *two states* $s_E = (s, mp_s, cp_s, ms_s, cs_s) \in S_E$, $s'_E = (s', mp_{s'}, cp_{s'}, ms_{s'}, cs_{s'}) \in S_E$, *a correct transition is a transition* $s_E \xrightarrow{l} s'_E \in T_E$ *such that* $cp = cp_s \cdot l$ *if* $cp_s \cdot l$ *is a prefix of* k, $cp = cp_s$ *otherwise, and* $|cp| + |cs_{s'}| \geq |k|$.

On the contrary, an *incorrect transition* is a transition that leads to a portion of the LTS where the sequence of actions k is never respected. We take into account the sum of the length of *mp* in the source state and of *ms* in its destination state. As for the correct transition, also in this case we need to take into account the label of the transition for the sum. If the sum is lower than the size of the k sequence the transition is classified as incorrect.

Definition 24 *(Incorrect Transition) Given an augmented LTS* $M_E^k = (S_E, s_E^0, \Sigma_E, T_E)$ *two states* $s_E = (s, mp_s, cp_s, ms_s, cs_s) \in S_E$, $s'_E = (s', mp_{s'}, cp_{s'}, ms_{s'}, cs_{s'}) \in S_E$, *an incorrect transition is a transition* $s_E \xrightarrow{l} s'_E \in T_E$ *such that* $mp = mp_s \cdot l$ *if* $mp_s \cdot l$ *is a prefix of* k, $mp = mp_s$ *otherwise, and* $|mp| + |ms_{s'}| < |k|$.

When a transition cannot be identified as correct nor as incorrect, it means that it is common to both correct and incorrect behaviours. Such transition is called a *neutral transition*.

The information concerning the detected types of transitions (correct, incorrect and neutral) is added to the augmented LTS in the form of tags. In this way we obtain the tagged LTS defined in Section 4.1, which can later be used to obtain neighbourhoods. Figure 6.8 depicts the tagged LTS obtained from the augmented LTS presented in Figure 6.2.

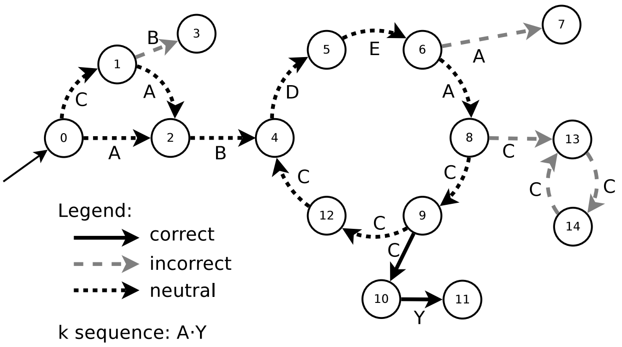


Figure 6.8: Transitions types computation.

Neighbourhood Example In Figure 6.9 we show again the example described in Figure 6.8, where the transition types computation has allowed to identify neighbourhoods (states coloured in grey). In particular state 9, where correct and neutral transitions are present, shows a neighbourhood of the first type, where a choice that will always satisfy the property is possible. On the contrary, states 1, 6 and 8 show neighbourhoods of the second type, where a choice that leads to an incorrect behaviour is possible.

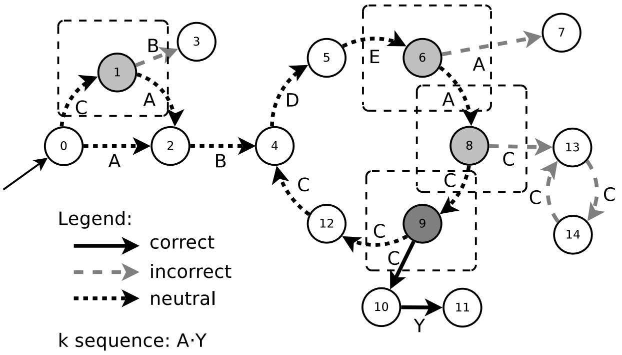


Figure 6.9: Neighbourhood identification.

6.4 Concluding Remarks

We comment here on some differences between the Prefix / Suffix approach and the Counterexample LTS one. First of all, note that both approaches generate a tagged LTS. However, the transitions types are computed in two different ways and rely on different definitions, since we retrieve them from two different models: the enriched counterexample LTS in the Counterexample LTS approach and the augmented LTS in the Prefix / Suffix one. While the former is a reduced model and contains only incorrect behaviours (correct transitions are directed to the sink state), the latter one contains both correct and incorrect behaviours.

Consequently, tagged LTSs produced with each approach have some specific characteristics. In a tagged LTS computed with the Counterexample LTS approach:

- correct transitions are only followed by the sink state;
- incorrect transitions are only followed by incorrect ones or by a final state;
- neutral transitions always precede correct and incorrect transitions.

Thus, a neutral transition tells us that we can still choose between a correct and an incorrect behaviour. Moreover, in this tagged LTS a path does not contain more than a correct transition, since all the correct behaviours are represented by the sink state.

In the case of a tagged LTS computed with the Prefix / Suffix approach neutral transitions can instead also follow correct and incorrect ones. Indeed, in this case the tagged LTS does not have a sink state to synthesize all the correct behaviours. Therefore, neutral transitions are not only precursors of a choice between correct and incorrect behaviours (as in the tagged LTS produced with the Counterexample LTS approach), but can also represent final parts of paths that are common to both behaviours. Let us consider as an example the portion of tagged LTS produced by the Prefix / Suffix approach in Figure 6.10. The depicted LTS splits in two branches representing respectively a correct and an incorrect behaviour, but at the end the two branches merge in a common suffix which is common to both behaviours. This suffix is thus composed of neutral transitions.

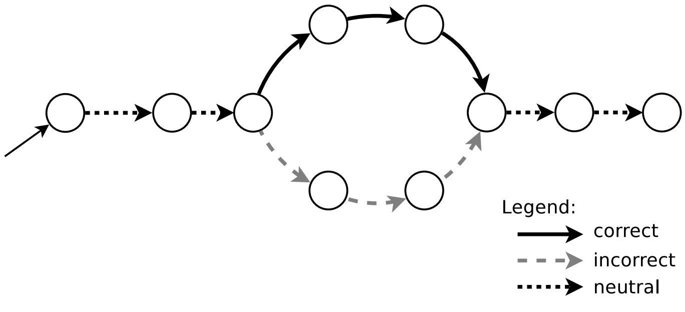


Figure 6.10: Portion of tagged LTS (refix / Suffix approach).

The different characteristics between the tagged LTSs produced with the two approaches do not have any influence on the neighbourhood detection. A neighbourhood is located only in states where an incoming neutral transition is followed by at least one outgoing correct or incorrect transition. Thus, the presence of incoming transitions of different types on a given state, which can only happen in a tagged LTS produced by the Prefix / Suffix approach, does not affect the neighbourhood detection. Therefore, the neighbourhood detection step is common to both approaches.

Finally, one of the perspective we have is to use the Prefix / Suffix approach with safety properties. We will discuss this in Chapter 9.

Chapter 7

Tool Support: the CLEAR Tool

In this chapter we present the implementation of our approach into the CLEAR tool, which is available online [cle]. The CLEAR tool architecture is depicted in Figure 7.1 and consists of three main modules: (i) neighbourhood calculation module, (ii) 3D visualization module and (iii) analysis module. The neighbourhood calculation module is responsible for the transition types recognition (implementing the Counterexample LTS and the Prefix / Suffix approaches) and for the neighbourhood computation. The result of this model is represented by the tagged LTS, which can be exploited by the 3D visualization module and by the analysis module. The latter one implements the abstraction techniques detailed in Chapter 4. We discuss implementation details for each module of the tool in the rest of this chapter.

7.1 CLEAR Neighbourhood Calculation Module

This module of the tool allows the computation of neighbourhoods, given a safety or a liveness property and a system specification. The core of this CLEAR module has been implemented in Java and consists of about 8000 lines of code. It also partially relies on the CADP toolbox [GLMS13], which enables one to specify and analyse concurrent systems using model and equivalence checking techniques.

To specify concurrent systems, we particularly make use of the LNT value passing process algebra [CCG$^+$18] and LOTOS [BB87]. Compilers provided by the CADP toolbox [GLMS13] are used to transform LNT and LOTOS specifications into LTS models, which are used as input format to our application. These LTS models are in BCG binary format or AUT ASCII format. In particular, the AUT one is used as intermediate format for the Java-based components of CLEAR. AUT files are then stored in memory using the Jung (Java Universal Network/Graph Framework) Java graph modelling library.

To specify temporal properties, we use the Mu-Calculus Logic (MCL) [MT08]. Liveness

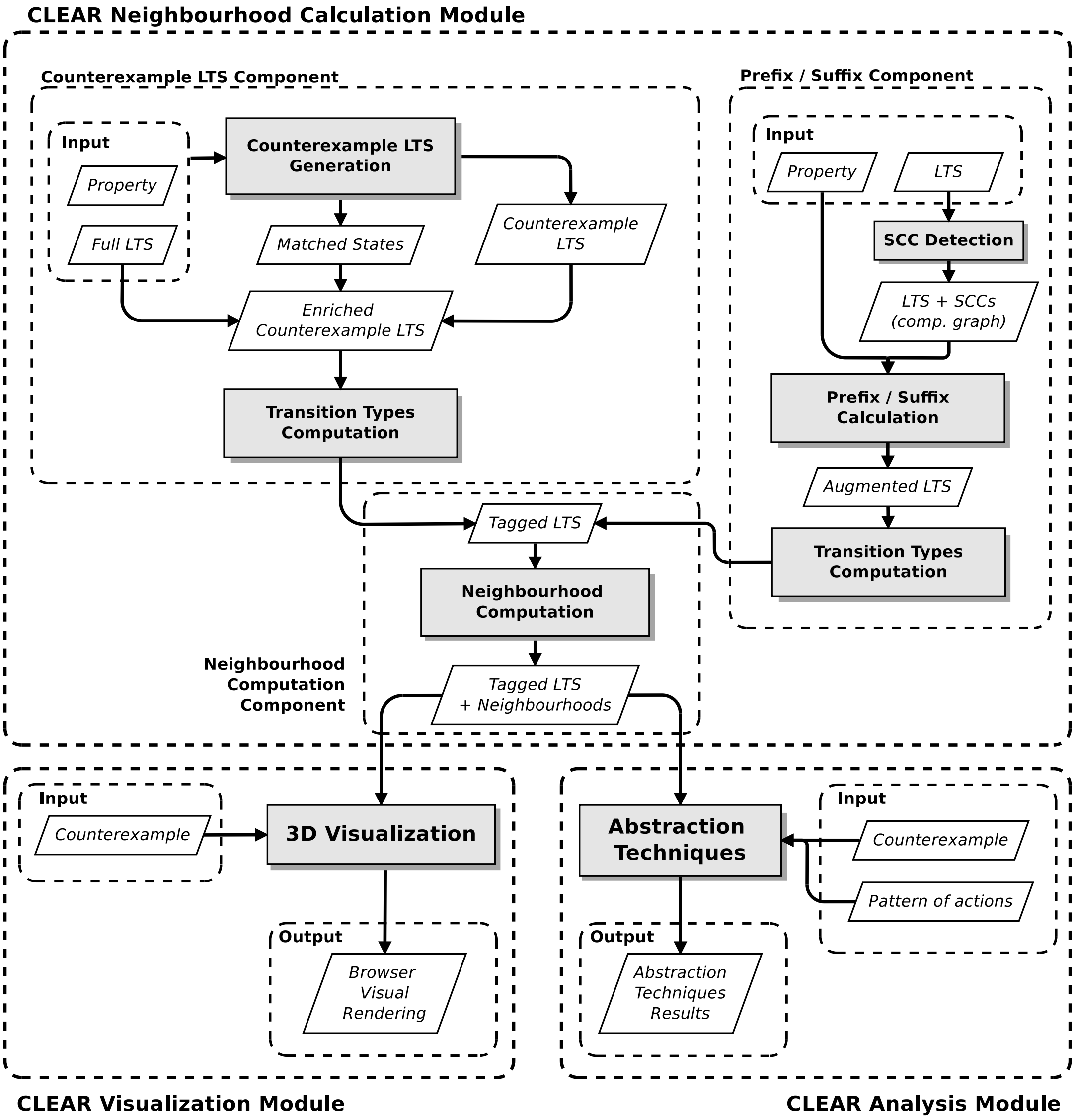

CLEAR Neighbourhood Calculation Module
Counterexample LTS Component
Input
Property
Full LTS
Counterexample LTS Generation
Matched States
Counterexample LTS
Enriched Counterexample LTS
Transition Types Computation
Prefix / Suffix Component
Input
Property
LTS
SCC Detection
LTS + SCCs (comp. graph)
Prefix / Suffix Calculation
Augmented LTS
Transition Types Computation
Tagged LTS
Neighbourhood Computation
Tagged LTS + Neighbourhoods
Neighbourhood Computation Component
Input
Counterexample
3D Visualization
Output
Browser Visual Rendering
Abstraction Techniques
Input
Counterexample
Pattern of actions
Output
Abstraction Techniques Results
CLEAR Visualization Module
CLEAR Analysis Module

Figure 7.1: Overview of the CLEAR tool modules.

properties are translated for simplicity in the form of sequences of strings representing the sequences of inevitable actions. To verify that an LTS respects a given temporal property, we make use of the CADP model checker (Evaluator [MT08]). Such model checker takes as input an MCL property and an input specification or LTS model, and returns a verdict (true or false + a counterexample if the property is violated).

The computation module is divided into three components. The first one implements the Counterexample LTS approach for computing the tagged LTS for safety property violations, The second one implements the Prefix / Suffix approach for computing the tagged LTS from liveness property violations. The third component allows computing neighbourhood in the tagged LTS, and is common to both approaches.

Counterexample LTS Component This component first allows the counterexample LTS generation step described in Chapter 5. The computation of the counterexample LTS is achieved by a script we wrote using SVL [GL01], a scripting language that allows one to interface with tools provided in the CADP toolbox. This script calls several tools: a specific option of Evaluator for building an LTS from a formula following the algorithm in [LM13]; EXP.OPEN for building LTS products; Reductor for minimizing LTSs; Scrutator [MPS12] for removing spurious traces in LTSs. After the generation of the counterexample LTS (in the form of AUT file) through the SVL script, the tool takes as input such model in order to perform the generation of the tagged LTS. The matching relation between states of the full and counterexample LTSs (obtained by the SVL script during the counterexample LTS generation) is then exploited to compare states of the two LTSs in order to extract correct transitions. Correct transitions are added as an attribute to transitions in the Jung model of the counterexample LTS, making it become an enriched counterexample LTS. In particular transition types (correct, incorrect and neutral) are defined as an enum type in the Java transition class. Correct transitions are later used to discover the neutral and incorrect ones in the enriched counterexample LTS, thus producing the tagged LTS.

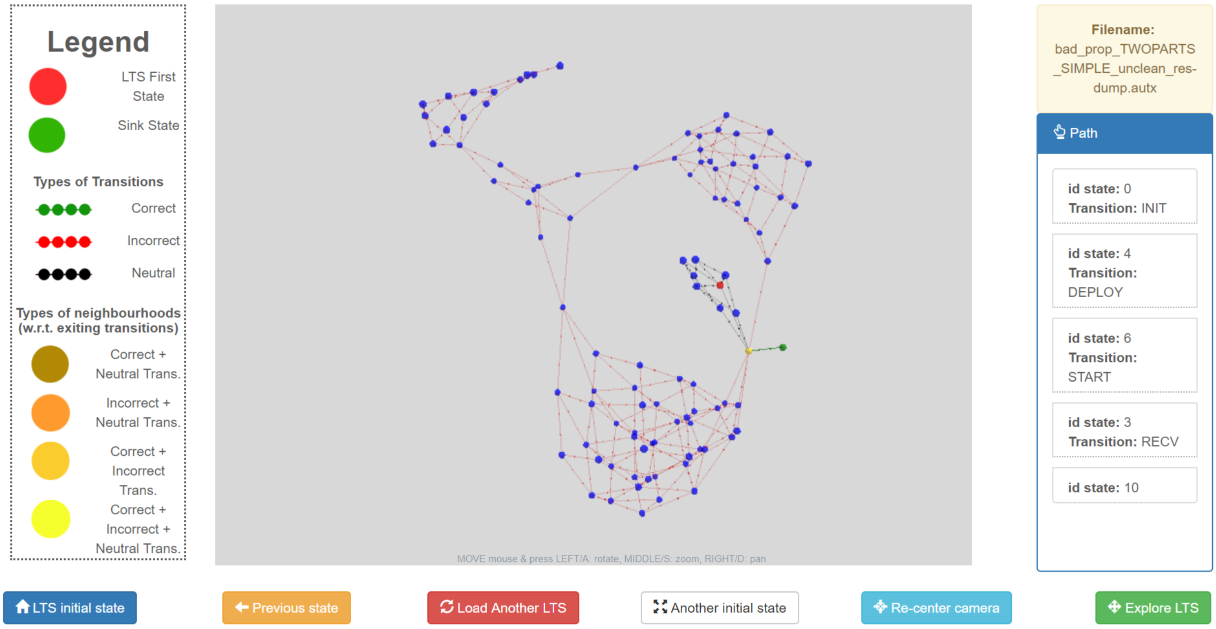
Prefix/Suffix Component This component of the tool implements the algorithms of the Prefix / Suffix approach detailed in Chapter 6. The sequence of inevitable actions describing the inevitability property is taken as input in the form of a sequence of strings. Prefix and suffix information is computed and stored on each state of the Java model. In particular we built some dedicated classes (CommonPref, CommonSuff, MaxPref and MaxSuff) to represent prefix and suffix objects, and we provide them as attributes of a state object. Transition types are later discovered and assigned to each transition using this information. A regression testing tool has also been implemented during the developmental phases of this component, in order to validate each improvement. A test set of 100 LTSs and sequences of inevitable actions have been built and used to support the regression tests.

Neighbourhood Computation Component Neighbourhoods are finally detected by analysing incoming and outgoing transitions of every state in the tagged LTS. The transition type (correct, incorrect or neutral) is assigned to the corresponding *transition* object instance through the graph modelling library. When a neighbourhood is detected, a specific identifier is added to the corresponding *state* object in the Java model of the LTS. Thus, the set of neighbourhoods is stored in the Java model of the tagged LTS.

7.2 CLEAR 3D Visualization Module

We present here the 3D visualizer, which supports the visualization of tagged LTSs with neighbourhoods. This component of the CLEAR tool has been developed as a web application using Javascript, the AngularJS framework, the bootstrap CSS framework and the 3D force graph library. An improved version of the AUT file format is used as input format for the 3D visualizer, and describes the tagged LTS with neighbourhoods. This component produces a 3D render of the tagged LTS where the neighbourhoods are highlighted with different colours, according to the neighbourhood taxonomy presented in Section 4.2.1. The developer can exploit various functionalities to better inspect the faulty model: forward/backward step-by-step animation, counterexample visualization, zoom in-/out on specific states or neighbourhoods, etc.

Figure 7.2 depicts a screenshot of the 3D visualizer in a web browser. One can see the different colours used in the LTS visualization with the legend on the left hand side. All functionalities appear in the bottom part of the figure. When the LTS is loaded, there is also the option to load a counterexample. On the right hand side, there is the name of the file and the list of states/transitions of the current animation. Note that transition labels are not shown, they are only displayed through mouseover. This choice allows the tool to provide a clearer view of the LTS.


Legend
● LTS First State
● Sink State
Types of Transitions
●●●● Correct
●●●● Incorrect
●●●● Neutral
Types of neighbourhoods (w.r.t. exiting transitions)
● Correct + Neutral Trans.
● Incorrect + Neutral Trans.
● Correct + Incorrect Trans.
● Correct + Incorrect + Neutral Trans.
MOVE mouse & press LEFT/A: rotate, MIDDLE/S: zoom, RIGHT/D: pan
Path
LTS initial state
Previous state
Load Another LTS
Another initial state
Re-center camera
Explore LTS

Filename: bad_prop_TWOPARTS_SIMPLE_unclean_res-dump.autx

id state: 0 **Transition:** INIT

id state: 4 **Transition:** DEPLOY

id state: 6 **Transition:** START

id state: 3 **Transition:** RECV

id state: 10

Figure 7.2: Screenshot of the CLEAR 3D visualization.

7.3 CLEAR Analysis Module

Finally, the last module of our tool provides the implementation of the abstraction techniques described in Section 4.3.2. First of all, note that the abstracted counterexample technique has been built in Java and relies on the CADP toolbox for the counterexample generation. It first produces the counterexample from an LTS and a property using the

Evaluator model checker. Second, it performs the counterexample reduction by locating and keeping actions that correspond to neighbourhoods, by comparing states of the counterexample to the ones belonging to the tagged LTS. The result retrieved by this technique consists of a counterexample abstracted in the form of a list of sub-sequences of actions, accompanied by the list of all neighbourhoods.

The other abstraction techniques presented in Section 4.3.2 (e.g., the shortest path to a neighbourhood and the pattern-based ones) have been developed using the Neo4j framework. We use the Neo4j graph database to store the tagged counterexample LTS and we built abstraction techniques as Neo4j queries. In this way the counterexample LTS enriched with the neighbourhoods behaves like a database that we can interrogate to obtain the desired information. Queries are built using the Cypher language, a graph query language developed for the Neo4j graph database. We used Neo4j for these abstraction techniques instead of using a custom-made Java solution since Neo4j allows an easier implementation of the pattern-based techniques, because of the use of a query language, and of the shortest path technique, thanks to the `shortestpath` function provided by the Cypher language. A translator class has been built in our main tool in order to translate an LTS from the Java Jung format to a Neo4j representation. The graph database is structured as follows. Nodes represent states and tags are used to classify the initial state, the sink state, final states and neighbourhoods. This categorisation allows multiple tags for the same state. For instance, a state can be tagged with the *state* tag, the *neighbourhood* tag and the *initialstate* tag at the same time. Neo4j relationships are used to represent transitions while properties are used to characterise the transition type (*correct*, *incorrect* and *neutral*) and the neighbourhood type (following the taxonomy introduced in Section 4.2.1). This database is then queried with the chosen abstraction technique. For instance, the shortest path from the initial node to a neighbourhood is retrieved with the Cypher query depicted in Figure 7.3.

```
MATCH (init:INITIALSTATE), (nb:NEIGHBOURHOOD),
path = shortestpath((init)-[*]->(nb))
RETURN path ORDER BY length(path) LIMIT 1
```

Figure 7.3: Shortest path query.

Another example of query is the one in Figure 7.4, which allows the retrieval of the shortest path to a neighbourhood through a given pattern. In this query, the pattern of actions is represented by `'.*.ACTION1.*.ACTION2.*.'`. The result of abstraction techniques is displayed through the Neo4j GUI.

```
MATCH (init:INITIALSTATE), (nb:NEIGHBOURHOOD),
path = shortestpath((init)-[*]->(nb))
WITH path, extract(n IN relationships(path) | n.action) AS actions
WITH path,
REDUCE (s = HEAD(actions), n IN TAIL(actions) | s + ’, ’ + n) AS result
WHERE result =∼ ’.*.ACTION1.*.ACTION2.*.’
RETURN path ORDER BY length(path) LIMIT 1
```

Figure 7.4: Shortest path to a neighbourhood through a given pattern query.

7.4 Concluding Remarks

We have seen in this chapter how each step of our approach has been implemented as a CLEAR tool module (neighbourhood calculation module, 3D visualization module and analysis module). In the next chapter, we apply our CLEAR tool to various real-world examples, making use of each module to debug faulty models.

Chapter 8

Experiments

This chapter presents the evaluation of our approach on real-world examples from various application areas. We divide the discussion of experiments in two sections, following the methodology presented in Section 4.3. Thus, we first detail experiments we carried on using our 3D visualization techniques. We show how the 3D visual rendering can be used to graphically observe the faulty model and see how neighbourhoods are distributed, allowing visual debugging of the model and identifying typical cases of bugs.

In the second section we focus on experiments using abstraction techniques. We first present some experiments on which we perform the annotation of transitions types in the model and the computation of neighbourhoods, with a quantitative point of view: size of the models, number of discovered neighbourhoods, computation time, etc. We discuss how the counterexample abstraction techniques can help in terms of reduction of the number of actions in counterexamples, highlighting only the relevant ones. We then detail the use of some abstraction techniques in four real-world case studies, taking into account both violations of safety and liveness properties, in order to show the benefits of such techniques in the debugging of faulty models. We finally present an empirical study that we put in place to validate our approach and abstraction techniques in collaboration with a set of developers.

8.1 3D Visualization Techniques Experiments

In this section, we show how our 3D visual rendering can be useful in order to better understand bugs. We present some experiments of faulty LNT specifications. Each specification is accompanied with a temporal property characterizing a requirement that is supposed to be satisfied by the specification. Model checking techniques are used and confirm that each property is violated by the corresponding specification. The LNT specifications we present in this section exhibit typical bugs inherent to concurrent systems, e.g., bugs related to parallel composition or to non-deterministic choices. Other interesting of bugged specifications are available online [cle], e.g., a model where the whole LTS is false (only red transitions), or a specification where the bug can be reached from a single neighbourhood, which represents a mandatory and unique choice to obtain the bug. Moreover, it is worth noting that these experiments are all generated by safety properties violations, so the tagged LTSs depicted in the figures are all built using the Counterexample LTS approach. One of the perspective we have is to apply the 3D visualization method also to tagged LTSs produced with the Prefix / Suffix approach. We will discuss about this in Chapter 9.

8.1.1 Interleaving Bug

The first example presents a simple specification given in Figure 8.1. This piece of code could be included in a larger LNT specification. The LNT process consists of a parallel construct with two branches. In the first branch, there is a sequence of actions EXECi. In the second branch, there is a choice between a null statement (correct termination) and a LOSS action. The safety property states that a LOSS action should never happen. This is written in MCL as follows:

```
( [ true* . 'LOSS' . true* ] false )
```

```
process Main [EXEC1, EXEC2, EXEC3, EXEC4, EXEC5, LOSS: none] is
   par
      EXEC1; EXEC2; EXEC3; EXEC4; EXEC5
   ||
      select
         LOSS
      []
         null
      end select
   end par
end process
```

Figure 8.1: LNT code for the interleaving bug.

The corresponding erroneous LTS computed and colored with the techniques presented in the former sections is given in Figure 8.2. The initial state is at the far left of this figure and appears in orange (not in red as is usually colored the initial state) because that state is a neighbourhood. This visualization shows that there is a sequence in which the bug does not appear (sequence of neutral black transitions leading to a correct green one). This happens when all EXECi actions execute and the second branch terminates correctly (null). Interestingly, we can see that at any state there is an incorrect transition (in red) corresponding to the execution of the LOSS action. This is typical of a bug which is interleaved with other actions, and the representation in our tool is similar to a *comb*.

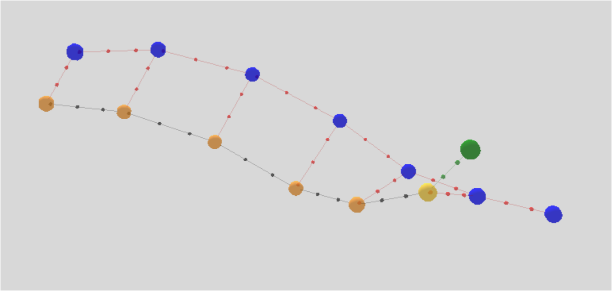


Figure 8.2: Visualization of the interleaving bug.

8.1.2 Interleaving Bug (V2)

The second example presents an extended and more complicated version of the former example. The LNT specification given in Figure 8.3 consists of three parts in sequence. The initial part (INITi actions) and the final part (CLOSEi actions) are used, respectively, for initialisation and closing purposes. The central part consists of a parallel composition where several EXECi actions are executed in parallel with another branch where there is a **select** construct. The **select** construct allows one to choose between two branches with several SENDi actions. The property states that a SEND2 action should never be followed by a SEND1 action:

([true* . 'SEND2' . true* . 'SEND1' . true*] false)

The erroneous LTS is given in Figure 8.4. The red state on the left hand part corresponds to the initial state. Then, we can clearly distinguish the initial part (left) with black transitions because all these transitions can lead to a possibly erroneous part of the system. Likewise, we can see on the right hand part of this figure the closing part of the specification where all transitions are incorrect (red) and where the bug cannot be avoided. These two parts (entirely black or entirely red) can be viewed as *noise* or actions that are not helpful from a debugging perspective. In contrast the central part of the figure is highly interesting. There are six neighbourhood states in that part of the LTS corresponding to a choice between executing a correct part of the specification (avoiding the sequence with a SEND2 action followed by a SEND1 action) leading to the white state (sink state), or executing an incorrect part of the specification. There are six choices because this choice is in parallel with the sequence of EXECi actions and can then appear at different states (interleaving). This is typical of a bug which is interleaved with other actions, looking in that case like a *spider web* due to the attraction of the sink state in the visualization.

8.1.3 Iteration Bug

This LNT specification (Fig. 8.5) exhibits a looping process with a nondeterministic choice executed at each iteration of that loop. In one of the two branches of the choice, there is a

```
process Main [ EXEC1, EXEC2, EXEC3, EXEC4, EXEC5, LOSS: none,
               INIT1, INIT2, INIT3: none,
               CLOSE1, CLOSE2, CLOSE3, CLOSE4: none,
               SEND1, SEND2, SEND3, SEND4: none] is

   (* initialisation part *)
   par
      INIT1 || INIT3; INIT1 || INIT1; INIT2
   end par;
   (* central part *)
   par
      EXEC1; EXEC2; EXEC3; EXEC4; EXEC5
   ||
      select
         par
            SEND2; SEND3 || LOSS
         end par;
         SEND1; SEND4
      []
         SEND1;
         par
            SEND2 || SEND2; SEND3 || LOSS
         end par;
         SEND4
      end select
   end par;
   (* closing part *)
   select
       par
           CLOSE3; CLOSE2 || CLOSE4; CLOSE1 || CLOSE2 || CLOSE1
       end par
    []
       CLOSE1; CLOSE2
   end select

end process
```

Figure 8.3: LNT code for the interleaving bug (V2).

parallel construct that allows one to obtain a LOSS action followed by a REC action, which is the sequence of actions that must not happen according to the following MCL property:

$$([\ \text{true*}\ .\ \text{'LOSS'}\ .\ \text{'REC'}\ .\ \text{true*}\ ]\ \text{false})$$

The visualization of the erroneous part of the LTS corresponding to this LNT specification looks like a *flower* and is given in Figure 8.6. Each petal corresponds to an iteration of the loop. There is a neighbourhood present at the beginning of each iteration, which represents a choice between reaching the incorrect behaviour, going to the sink state (both at the center of the picture), or continuing to the next petal. All the petals consist of

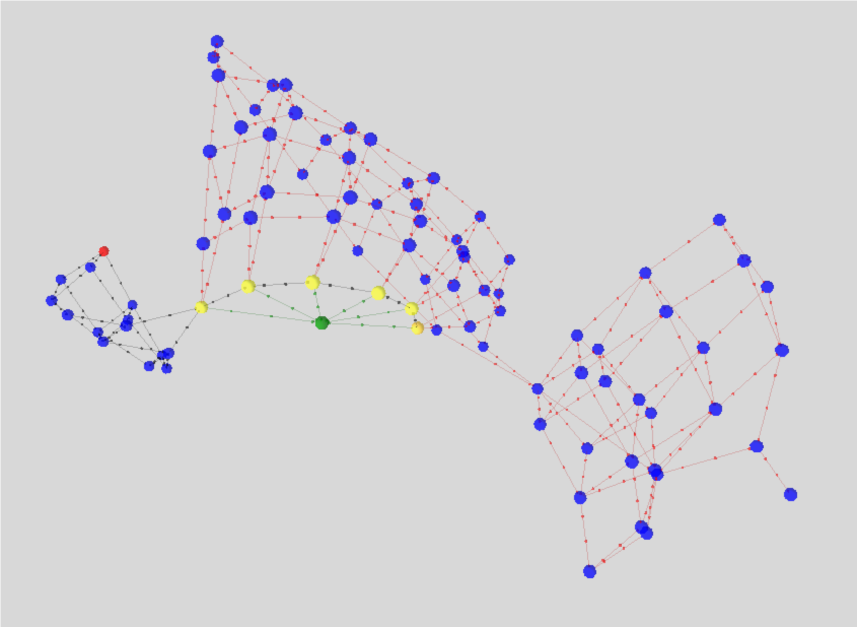


Figure 8.4: Visualization of the interleaving bug (V2).

```
process Main [WAIT, INIT, REC, EXEC1, EXEC2, LOSS, STORE: none] is
   var I,K : nat in
      I := 0;
      K := 10;
      for I:=0 while I<K by I:=I+1 loop
         WAIT;
         select
            par
               EXEC1; STORE || LOSS
            end par
         []
            par
               REC; EXEC2 || LOSS
            end par;
            I:=10
         end select
      end loop
   end var
end process
```

Figure 8.5: LNT code for the iteration bug.

neutral (black) transitions because the bug can still be avoided. There is a part of the LTS with red transitions, which is reached after executing an incorrect transition in one of the aforementioned neighbourhoods. After nine iterations, executing at each iteration the first branch of the select construct, a final correct transition leads to the sink state and makes the whole specification definitely avoid the incorrect part of the behaviour.

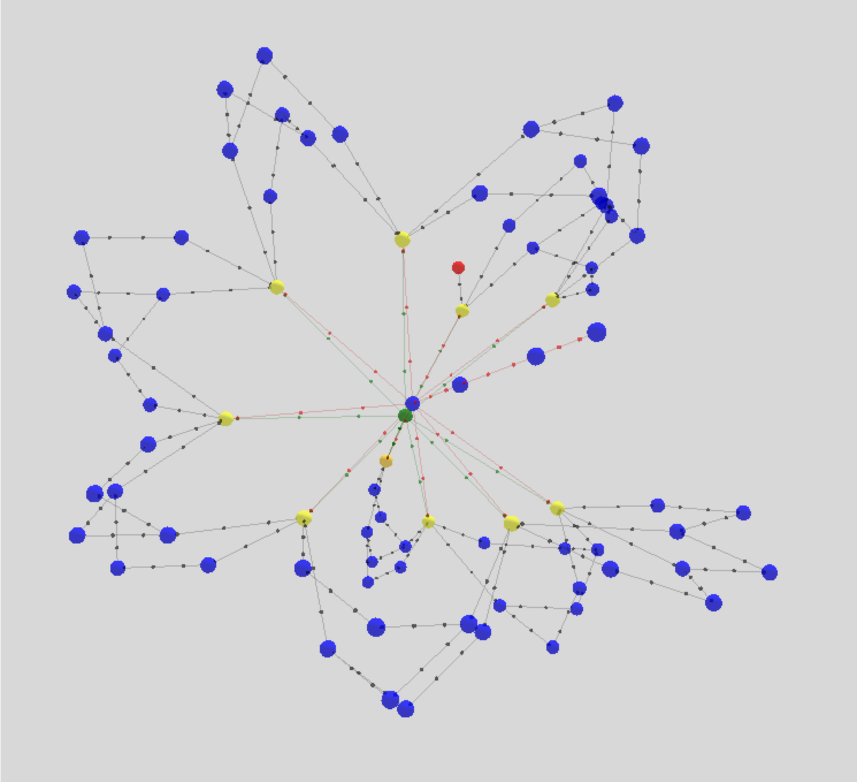


Figure 8.6: Visualization of the iteration bug.

8.1.4 Causality Bug

The fourth example is a producer-consumer system. The LNT specification consists of about 100 lines of code and is available online [cle]. The specification is composed of three main processes: a producer process, a consumer process and a process that can either be a consumer or a producer. This last process is given in Figure 8.7. Each process can loop infinitely or break the loop and terminate the execution. A deployer process is also part of the specification in order to initiate the three other processes. The provided property states that a process cannot consume if something has not been produced before. This is written in MCL as follows:

```
[ ( not "PRODUCE" )* . "CONSUME" . true* ] false
```

The specification violates this property when the PRODCONS process (Fig. 8.7) acts as a consumer, because it can consume without ensuring that PRODUCE has been performed beforehand.

```
process PRODCONS [ CONNECT, READY, SYNC, WAIT : none,
                   DEPLOY, START, IAMPRODUCER, IAMCONSUMER : WHOAMI_C,
                   CONSUME, PRODUCE : none
                 ] is
  var whoami : bool in
    DEPLOY(1 of nat);
    START(1 of nat);
    CONNECT;
    select
      whoami := true; IAMPRODUCER(1 of nat)
    []
      whoami := false; IAMCONSUMER(1 of nat)
    end select;
    WAIT;
    READY;
    if (whoami) then
     loop L in
        select
          NULL [] break L
        end select;
        par
          WAIT || PRODUCE; SYNC
        end par
      end loop
    else
      loop L in
        select
          NULL [] SYNC [] break L
        end select;
        par
          WAIT || CONSUME
        end par
      end loop
    end if
  end var
end process
```

Figure 8.7: LNT code for the causality bug.

The erroneous LTS with colored transitions and neighbourhoods is given in Figure 8.8. The LTS is divided into three parts. The initial part represents the portion of code in which every process performs the deployment and this part of the model has no impact on the bug (no neighbourhoods and all neutral transitions). Then, a set of neighbourhoods of the same type is present between the first part of the LTS and the second (central) one. These neighbourhoods have all a correct and a neutral transition, and represent the first choice that contributes to the cause of the bug (when the PRODCONS process decides to be a consumer). Those neighbourhoods can be viewed as a *frontier* between the initial and central part of the LTS. All the correct transitions are directed to the sink state, that abstracts the correct part of the LTS. A second frontier is present between the central part of the LTS and the third part (with all red transitions). This frontier is composed of neighbourhoods that represent the second cause of the bug, that happens when a CONSUME action has been performed without an initial PRODUCE action. The figure with the two frontiers helps in understanding that there is a causality between both kinds of neighbourhoods. This is because neighbourhoods of the second frontier are reached only when a neutral transition that represents the choice to be a consumer is taken in one of the neighbourhoods belonging to the first frontier.

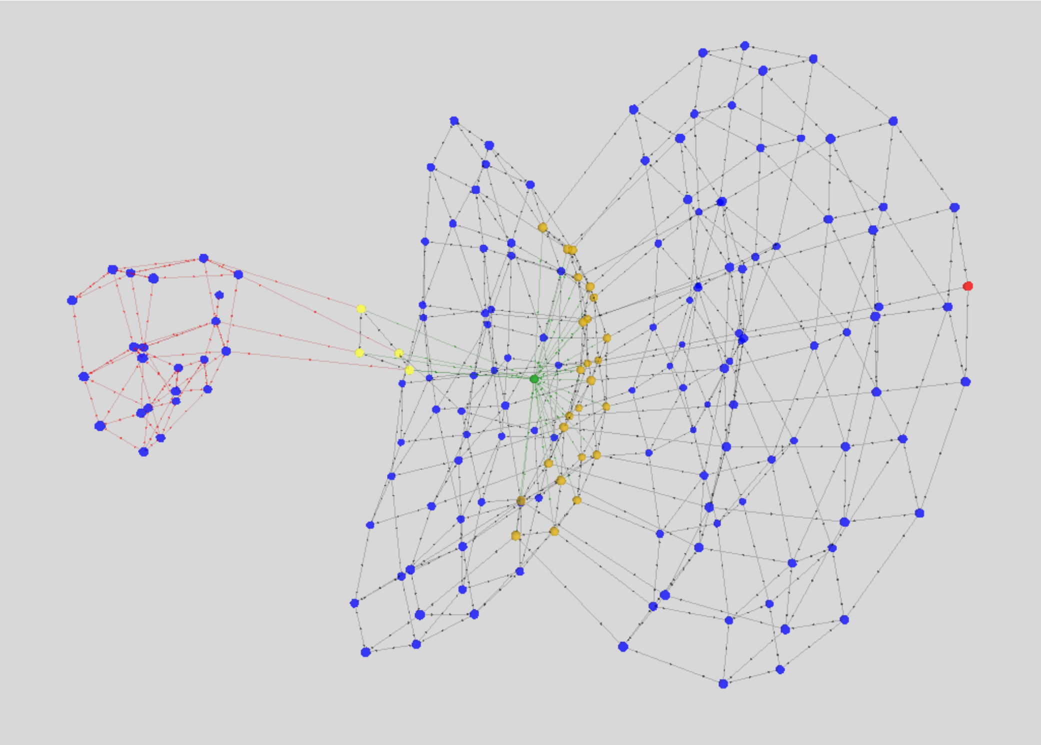


Figure 8.8: Visualization of the causality bug.

8.2 Abstraction Techniques Experiments

We present in this section some experiments we carried out on about 100 real-world examples. We first discuss our approach with a quantitative analysis on those experiments. Second, we apply our abstraction techniques to four real-world case studies, taking into account both violations of safety and liveness properties. Finally, we will present an empirical evaluation we built using the abstracted counterexample technique.

8.2.1 Quantitative Analysis

We now discuss our approach on various experiments with a quantitative point of view. For each experiment, we use as input an LNT specification or an LTS model, and an MCL property. Table 8.1 summarizes the results for some of these experiments, computed for safety properties violations. The first column contains the name of the model and a reference to a published article (when it exists). The second and third columns show the size of the full and the counterexample LTSs, respectively, in terms of number of states, transitions and labels. The fourth column gives the number of neighbourhoods for each neighbourhood type according to the neighbourhood taxonomy (*cn*: with only correct transitions, *in*: with only incorrect transitions, *ci*: with correct and incorrect transitions, *cni*: with correct, incorrect and neutral transitions). The fifth column presents the ratio of neighbourhoods over the number of states in the counterexample LTS as a percentage. We also present in the table the results of the counterexample abstraction technique, in terms of size of the shortest (retrieved with BFS exploration of the LTS) and of the abstracted counterexample, respectively. Finally, the last column details the total computation time (in seconds) for each test, which takes into account the counterexample LTS production, the transition types annotation and the neighbourhood detection.

First of all, one can note that there exist some particular cases in which there is only one neighbourhood in the model. This happens when the analysed property is always false. In these particular cases, we identify the initial state as a neighbourhood with only incorrect outgoing transitions. An example is represented by the *Peterson* algorithm case study, in which we have introduced a bug, and that we have verified with a property that guarantees mutual exclusion. Figure 8.9 shows a portion of the tagged LTS with the first (and sole) discovered neighbourhood, with only incorrect transitions outgoing from it. A traditional approach, where the developer analyses a single counterexample, would not have been able to let the developer understand that the bug always occurs in all the states of the system, and would have forced the developer to spend time trying to understand which actions were relevant in the counterexample, which in this case is worthless. The tagged LTS with the initial neighbourhood allows to understand that the bug is present in all the executions of the systems, meaning that all the possible executions are not correct, and to avoid an useless analysis of relevant actions. On the contrary, when the property is always verified in all the system executions (thus no counterexample is produced) the counterexample LTS is not generated and no neighbourhood is found (see the *restaurant booking* case).

Second, note that the average of the ratio between the number of neighbourhoods and the number of states in the counterexample LTS is around 10%. In one case (*multiway rendezvous (simple)*) that value is higher than 50%, but it is worth noting that in this case the specification of the system was modified to obtain a simplified version of the example.

Third, we applied the counterexample abstraction technique to the case studies presented in Table 8.1. We can see a clear gain in length between the original counterexample and the abstracted one, which keeps only relevant actions using our approach and thus

Example	L_F (s/t/l)	L_C (s/t/l)	*cn/in/ci/cin*	*ratio*	$\lvert Ce\rvert$	$\lvert Ce_r\rvert$	*time*
1. sanitary agency (v.1) [SBS04]	227 / 492 / 31	226 / 485 / 31	6 / 10 / 0 / 0	7.08	14	2	7.6s
2. sanitary agency (v.2) [SBS04]	142 / 291 / 31	526 / 1,064 / 31	12 / 10 / 4 / 2	5.32	64	7	7.7s
3. sanitary agency (v.3) [SBS04]	91 / 172 / 31	55 / 95 / 23	5 / 5 / 2 / 0	21.82	19	6	7.9s
4. SSH protocol (v.1) [MP11]	23 / 25 / 23	24 / 24 / 19	1 / 0 / 1 / 0	8.33	14	3	7.6s
5. SSH protocol (v.2) [MP11]	23 / 25 / 23	40 / 40 / 19	3 / 0 / 1 / 0	10.00	30	7	7.7s
6. client supplier (v.1) [CMS+10]	35 / 45 / 26	29 / 33 / 24	2 / 0 / 1 / 0	10.34	18	5	7.6s
7. client supplier (v.2) [CMS+10]	35 / 45 / 26	25 / 25 / 24	3 / 0 / 1 / 0	16.00	19	6	7.7s
8. client supplier (v.3) [CMS+10]	35 / 46 / 26	33 / 41 / 24	1 / 2 / 1 / 0	12.12	16	4	7.9s
9. train station [SBR12]	39 / 66 / 18	26 / 34 / 18	0 / 2 / 1 / 0	11.54	6	2	7.9s
10. selfconfig [SEP+13]	314 / 810 / 27	159 / 355 / 27	24 / 15 / 1 / 5	28.30	14	2	7.8s
11. CFSM [JJ93]	1,321 / 2,563 / 7	3,655 / 7,246 / 7	12 / 205 / 2 / 0	5.99	7	2	7.8s
12. online stock broker [FBS04]	1,331 / 2,770 / 13	3,516 / 7,326 / 13	44 / 145 / 17 / 0	5.86	23	2	7.9s
13. multiway rendezvous (simple) [EL17]	2,171 / 5,098 / 53	171 / 283 / 36	89 / 2 / 1 / 1	54.39	47	38	8.2s
14. multiway rendezvous [EL17]	1,318 / 3,217 / 53	539 / 1,186 / 41	143 / 8 / 4 / 4	29.50	47	22	7.7s
15. shifumi (2 players)	25 / 57 / 27	60 / 130 / 27	6 / 4 / 5 / 0	25.00	7	4	7.7s
16. shifumi (3 players)	193 / 690 / 67	499 / 1,814 / 67	30 / 53 / 10 / 18	22.24	7	2	7.9s
17. shifumi (4 players)	1,362 / 7,209 / 125	3,577 / 19,233 / 125	206 / 418 / 15 / 188	23.12	7	2	10.6s
18. shifumi (5 players)	9,693 / 70,596 / 201	25,593 / 188,466 / 201	1,622 / 2,999 / 20 / 1,598	24.38	7	2	12.2s
19. Peterson algorithm (v.1) [Pet81]	352,112 / 552,848 / 85	673,569 / 1,058,113 / 85	0 / 1 / 0 / 0	0.00	35	1	24.2s
20. Peterson algorithm (v.2) [Pet81]	352,112 / 552,848 / 85	683,958 / 1,074,948 / 85	0 / 1 / 0 / 0	0.00	35	1	24.1s
21. restaurant booking [MPS08]	55 / 78 / 31	-	0 / 0 / 0 / 0	0.00	0	0	7.2s
22. travel agency [SEG10]	60 / 99 / 26	60 / 99 / 26	0 / 1 / 0 / 0	1.67	22	1	7.6s
23. FTP transfer [BBC05]	49 / 86 / 18	45 / 76 / 18	0 / 7 / 1 / 2	22.22	10	2	7.6s
24. news server [OSB14]	21 / 34 / 8	14 / 18 / 6	0 / 1 / 1 / 2	28.57	4	2	7.7s
25. mars explorer [BP06]	52 / 74 / 34	45 / 66 / 26	0 / 0 / 1 / 0	2.22	23	2	7.6s
26. factory job manager [BFF09]	30 / 45 / 14	18 / 21 / 14	2 / 0 / 1 / 0	16.67	6	4	7.9s
27. vending machine [GMW12]	17 / 19 / 16	13 / 13 / 12	0 / 0 / 1 / 1	15.38	9	3	7.5s
28. restaurant service [vdAMSW09]	25 / 37 / 12	23 / 33 / 12	0 / 1 / 1 / 0	8.70	9	2	7.4s
29. CFSM (estelle) [JJ93]	4,058 / 8,219 / 9	24,171 / 50,830 / 9	1,344 / 973 / 16 / 9	9.69	18	2	8.5s
30. reactive system [LSW08]	266 / 720 / 5	296 / 786 / 5	0 / 1 / 0 / 0	0.34	3	1	7.8s
31. bug repository [GS11] report	80 / 158 / 13	108 / 203 / 13	0 / 1 / 0 / 0	0.93	15	1	7.6s
32. message exchange	666 / 2,281 / 20	1,166 / 3,857 / 20	0 / 74 / 0 / 74	12.69	17	2	7.9s
33. TFTP/UDP protocol [GT09] (v.1)	4 / 7 / 2	5 / 11 / 2	0 / 1 / 0 / 0	20.00	1	1	7.7s
34. TFTP/UDP protocol [GT09] (v.2)	98,205 / 9,018,043 / 11	214,217 / 19,852,120 / 11	2,957 / 2,444 / 1 / 16	2.53	3	3	558.3s
35. TFTP/UDP protocol [GT09] (v.3)	61,008 / 6,328,658 / 11	123,983 / 12,318,353 / 11	6,323 / 5,758 / 0 / 1,556	11.00	8	8	301.5s
36. TFTP/UDP protocol [GT09] (v.4)	98,205 / 9,018,043 / 11	214,217 / 19,852,616 / 11	2,955 / 2,444 / 1 / 16	2.53	3	3	571.5s

Table 8.1: Experimental results.

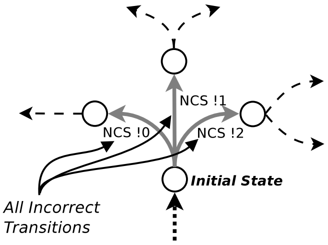


Figure 8.9: Peterson Algorithm: the discovered neighbourhood.

facilitates the debugging task for the user. For instance, cases in which the abstracted counterexample contains only two actions, like the *train station* case study, mean that we identified only one neighbourhood in the shortest counterexample. The *restaurant booking* example has both the original and the abstracted counterexample of length zero, since the model is correct (thus it does not produce a counterexample). There exist some particular cases in which the abstracted counterexample contains only one action. These are the cases in which there exists only one neighbourhood in the model, placed in the initial state, e.g., in the *Peterson* algorithm case study. Note that the abstracted counterexample technique in some cases does not work. As an example, note that in the case of the TFTP/UDP protocol, the abstracted counterexample technique has not reduced the size of the shortest counterexample. This occurs because the counterexample does not contain any neighbourhood. The last state reached in the counterexample LTS contains a final transition, and neutral transitions. In this case, it is the final transition itself which is incorrect.

Finally, as far as computation time is concerned, the table show that the time is quite low for small examples, while it tends to increase w.r.t. the size of the LTS when we deal with examples with hundreds of thousands of transitions and states. Note that an important part of the total time in examples with large LTSs is spent in loading LTSs on the system memory in the form of Jung graphs. The loading time takes about 30% of the total time for examples with about a million of states, while it remains negligible for small examples (about 1% of the computation time when dealing with LTSs with hundreds of states). A relevant part of the total time is also represented by the time for the counterexample LTSs production, which is slightly longer than the time for computing transitions types and neighbourhood. Indeed, while for very small examples the counterexample LTS production takes 98% of the total time, for average size ones (with thousands of states) this value decreases to 45% and reaches 5% in examples with millions of states. This is because the script involved in the counterexample LTS computation calls several CADP tools in sequence. Note that when we increase the size of the examples, dealing with hundreds of thousands of transitions and states, the time needed for the transitions and neighbourhood computation steps becomes predominant w.r.t. the one needed for the computation of the counterexample LTS.

8.2.2 Case Studies

We now present in detail four case studies: (i) the rock-paper-scissors game (shifumi) case study, (ii) the sanitary agency case study, (iii) the alternating bit protocol case study and (iv) the multiway rendezvous protocol case study.

Case Study: Shifumi Tournament

The *shifumi* case study models in LNT a tournament of rock-paper-scissors games. In a typical game between two players each player forms one of the three possible shapes (rock, paper or scissors). Each shape allows defeating one of the two others, but is defeated by the remaining one (e.g. the rock defeats the scissors but is defeated by the paper). When a shape is used against the same shape the game ends in a draw, and it is repeated. The tournament allows more than two players to compete. When a player wins a game, she continues playing with the next player. On the contrary, the player who loses the game has to stop playing. The tournament continues until there is only a winner. Figure 8.10 shows the LNT process for a player.

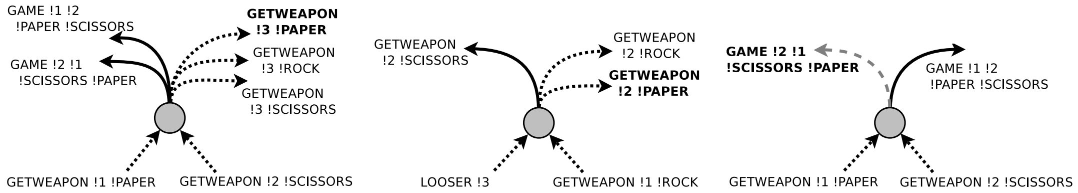
```
process player [GETWEAPON: getweapon, GAME: game,
                LOOSER: nat] (self: nat, honest: bool) is
  var opponent: nat, mine, hers: weapon in
    loop
      GETWEAPON (self, ?mine);
      select
        GAME (self, ?opponent, mine, ?hers)
      []
        GAME (?opponent, self, ?hers, mine)
      end select;
      if wins_over (mine, hers) then
        LOOSER (opponent)
      elsif wins_over (hers, mine) then
        if (not (honest)) and (mine == rock) then null
        else stop
        end if
      end if
    end loop
  end var
end process
```

Figure 8.10: LNT code for a shifumi player.

We discuss here one of the shifumi tournaments case studies described in Table 8.1, which represents a tournament between three players. A safety property is provided to guarantee no cheating by any player. More precisely, it avoids that a player who has previously lost can play again in the tournament, and it is written in MCL as follows:

```
[ (true* . 'LOOSER !1' . true* . 'GAME .*. !1 .*'. true*) |
(true* . 'LOOSER !2' . true* . 'GAME .*. !2 .*'. true*) |
(true* . 'LOOSER !3' . true* . 'GAME .*. !3 .*'. true*) ] false
```

The analysed tournament contains a bug, since a dishonest player is able to play again after having lost a game. We have generated the counterexample LTS and applied two abstraction techniques: the abstracted counterexample technique (to a randomly produced counterexample), and the shortest path to a neighbourhood technique. The abstracted counterexample reduces the number of actions from 14 to 9, since it involves 5 neighbourhoods in the counterexample LTS. The detected neighbourhoods precisely identify the origin of the bug. Indeed they allow us to understand that player one is the cheater and that it cheats after having lost a game using *rock* as shape. First, the initial neighbourhood indicates that alternative combinations that avoid the occurrence of the bug are possible (see Figure 8.11a). Second, a subsequent neighbourhood highlights a game *rock* versus *paper* between player one and player two, where player one loses and thus should have exited the tournament (see Figure 8.11b). This neighbourhood is interesting also because it shows that if player two chooses *scissors* she would lose that game and the bug would not occur. Finally, the last neighbourhood points out that a new game is played between player one and two, but this should not be possible, since the previous neighbourhood has indicated that player one has lost (see Figure 8.11c).



(a) (b) (c)

Figure 8.11: Three of the neighbourhoods detected in the abstracted counterexample.

The shortest path to a neighbourhood technique (depicted in Figure 8.12) shows that the discovered neighbourhood, which is only two transitions far from the initial state, belongs to the first type (outgoing correct and neutral transitions). The correct transitions show games between players one and two using *scissors* and *paper*, while the neutral ones show the selection of the three possible shapes by player three. This means that the use of *scissors* and *paper* shapes between players one and two avoids the bug, and confirms that player one must use a *rock* to cheat. Moreover, neutral transitions show that the choice of the shape by player three has no impact on the bug. The combined use of the two abstraction techniques allowed us to have a finer information about the bug.

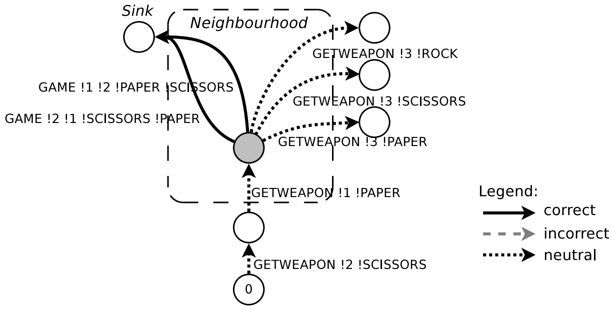


Figure 8.12: Shortest path to a neighbourhood.

Case Study: Sanitary Agency

We now describe the *sanitary agency* [SBS04] example (see the third line in Table 8.1), which models an agency that aims at supporting elderly citizens in receiving sanitary assistance from the public administration. The model involves four different participants: (i) a citizen who requests services such as transportation or meal; the request can be accepted or refused by the agency; (ii) a sanitary agency that manages citizens' requests and provides public fee payment; (iii) a bank that manages fees and performs payments; (iv) a cooperative that receives requests from the sanitary agency, receives payments from the bank, and provides transportations and meal services. Figure 8.13 gives the LTS model for each participant. We assume in this example that the participants interact together asynchronously by exchanging messages via FIFO buffers.

We evaluated two different properties on this case study, a liveness and a safety one.

Sanitary Agency: Liveness Property

We present a liveness property with two nested inevitable executions. The property states that the treatment of a citizen request by the agency (represented by the `REQ_EM` action) should always take place, and should always be followed by the reception of a transport service by the citizen (represented by the `PROVT_REC` action). Such a property is written in MCL as follows:

$$\begin{gathered}\mathsf{mu\ X\ .\ ((<\ ("REQ\_EM")\ >\ true\ or\ <\ not\ ("REQ\_EM")\ >\ X)\ and\ [\ ("REQ\_EM")\ ]}\\ \mathsf{(mu\ X\ .\ ((<\ ("PROVT\_REC")\ >\ true\ or\ <\ not\ ("PROVT\_REC")\ >\ X)}\\ \mathsf{and\ [\ not\ ("PROVT\_REC")\ ]\ X))\ and\ [\ not\ ("REQ\_EM")\ ]\ X)}\end{gathered}$$

Our tool identifies five neighbouroods in the model. We then apply the abstracted counterexample technique to the shortest counterexample, allowing to discover two neighbourhoods and consequently reducing the length of the counterexample from 15 actions to 4. The top side of Figure 8.14 depicts the shortest counterexample while the bottom side depicts the corresponding neighbourhoods. The extracted actions are relevant since the neighbourhoods to which they belong precisely identify choices in the model that violate

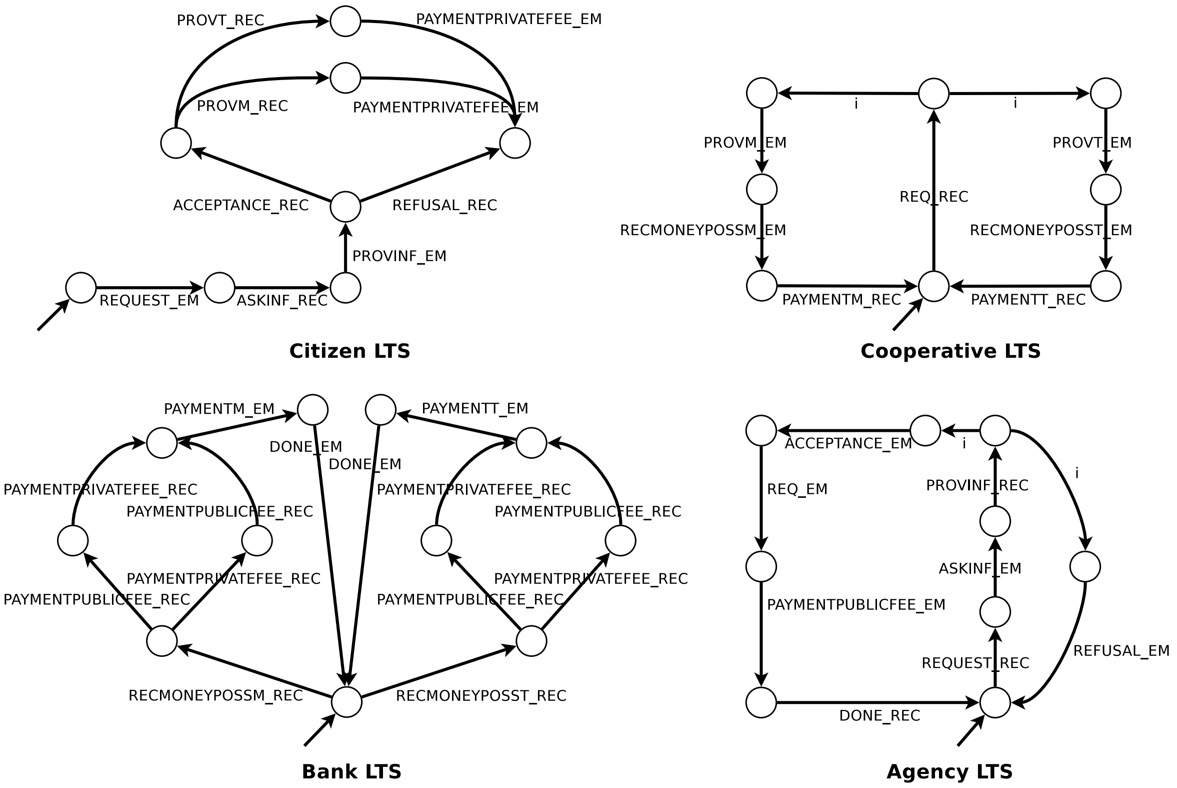


Figure 8.13: LTS models for the sanitary agency.

the property. In this case, the first neighbourhood shows that the first action in the property is not inevitable. The REQ_EM action can take place only after an ACCEPTANCE_EM action, but the neighbourhood exhibits an incorrect transition with the REFUSAL_EM action, revealing that the citizen request can be refused and thus preventing its treatment. The second neighbourhood shows that, even when the citizen request is treated by the agency, the system does not always satisfy the nested inevitable action, since it can also provide meal services. This is highlighted by the choice between the correct transition with the PROVT_EM action (emission of a transport service) and the incorrect transition with the PROVM_EM action (emission of a meal service).

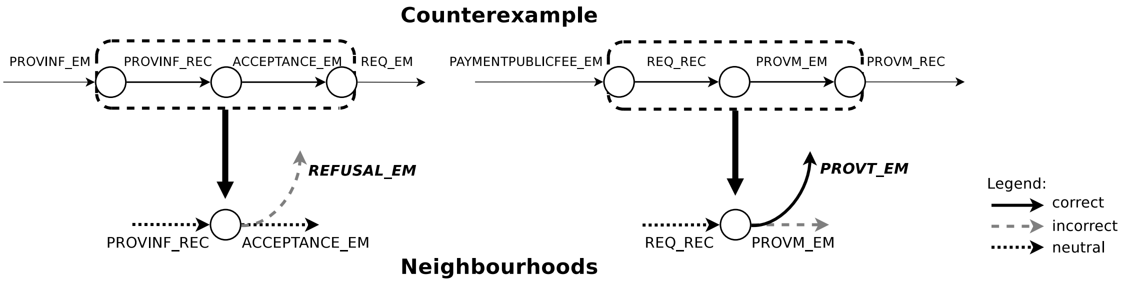


Figure 8.14: Sanitary agency: shortest counterexample and neighbourhoods (liveness).

Sanitary Agency: Safety Property

For illustration purposes, we use an MCL safety property, which indicates that the payment of a transportation service to the transportation cooperative cannot occur after submission of a request by a citizen to the sanitary agency:

[true* . 'REQUEST_EM' . true* . 'PAYMENTT_EM' . true*] false

Our tool was able to identify twelve neighbourhoods in the counterexample LTS, divided into five neighbourhoods from correct transitions, five from incorrect transitions and two from correct and incorrect transitions (without neutral transitions).

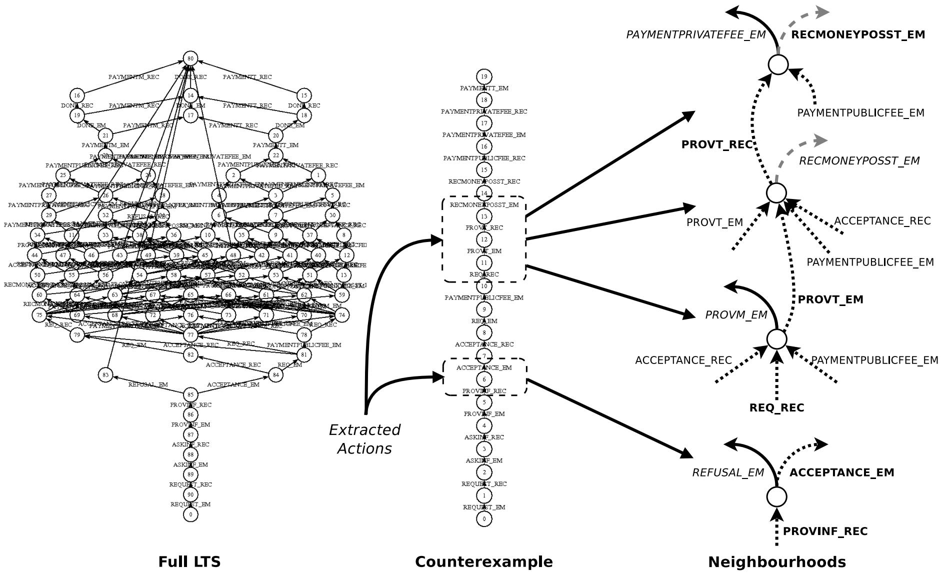


Figure 8.15: Sanitary agency: full LTS and shortest counterexample (safety).

We applied the abstracted counterexample technique to the shortest counterexample. The shortest counterexample involves four neighbourhoods, and this allows us to reduce its size from 19 actions to only 6 actions. Figure 8.15 shows (from left to right) the full LTS of the sanitary agency model, the shortest counterexample, and the four neighbourhoods for this counterexample. Actions that appear in the counterexample are highlighted in bold. The neighbourhoods and corresponding extracted actions are relevant in the sense that they precisely identify choices that lead to the incorrect behaviour. In particular, they identify the two causes of the property violation and those causes can be observed on the shortest counterexample. The first cause of violation is emphasized by the first neighbourhood and occurs when the citizen request is accepted. In that case, the refusal of the request is a correct transition and leads to a part of the LTS where the property is not violated. The three next neighbourhoods pinpoint the second reason of property violation. They show that actions that trigger the request for payment of the transportation services have been performed, while this is not permitted by the property. Note that different neighbourhood types provide us with different information about choices, according to the neighbourhood taxonomy. Finally, note that in this specific case the abstracted counterexample technique was sufficient to understand the cause of the bug.

Case Study: Alternating Bit Protocol

We now discuss the *Alternating Bit Protocol* case study, which consists of a data link layer network protocol that allows the retransmission of lost or corrupted messages. The version of the protocol analysed here, available as CADP demo [Inr], is a variant without data values written in LOTOS (the reader interested in LOTOS can refer to [BB87] for more details). The model is composed of four processes: a TRANSMITTER process that acquires and sends a message; a RECEIVER process that gets a message; MEDIUM1 and MEDIUM2 processes that represent transmission channels. The demo is provided with an inevitable execution property that states that a PUT action will be eventually reached from the initial state and which is written in MCL as follows:

$$\text{mu X . (< true > true and [not "PUT"] X) false}$$

This property is not satisfied by the model because of the presence of loops in the specification that can lead to an infinite trace that never reaches a PUT action. More precisely, the problem is caused by an interaction between the RECEIVER and the MEDIUM2 processes, depicted in Figure 8.16. When the TRANSMITTER process has not yet started the message treatment (represented by the PUT action), the receiver might have to wait. In this case the RECEIVER produces a TIMEOUT action, followed by the sending of an incorrect ack message (RACK1 action). If this ack message is lost by MEDIUM2 (LOSS action) and the receiver is still waiting, a loop might be produced until the TRANSMITTER starts the message treatment.

Our tool detects six neighbourhoods in the model, all with correct and neutral transitions. In this case the use of the counterexample abstraction technique is not useful, since each action in the counterexample belongs to a neighbourhood. The shortest path to a neighbourhood technique may partially help, but it would not return any path, because a neighbourhood exists in the initial state. However, in this case our neighbourhood notions can help in understanding the bug even without the use of abstraction techniques. Figure 8.17 depicts a portion of the model with these neighbourhoods, which are located at states 0, 2, 5, 12, 13 and 23. In particular, neighbourhoods at states 5, 12, 13, 23 present choices that make the execution of the system remain infinitely inside the loops, preventing the satisfaction of the property by reaching the PUT action. This is highlighted by neutral transitions containing LOSS, RACK1 and TIMEOUT actions that repeat the cycle.

```
process MEDIUM2 [RACK0, RACK1, SACK0, SACK1, SACKe] : noexit :=
   RACK0;          (* acknowledge reception *)
   (
      SACK0;       (* correct transmission *)
      MEDIUM2 [RACK0, RACK1, SACK0, SACK1, SACKe]
   []
      SACKe;       (* loss with indication *)
      MEDIUM2 [RACK0, RACK1, SACK0, SACK1, SACKe]
   []
      (hide LOSS in
      LOSS;        (* silent loss *)
      MEDIUM2 [RACK0, RACK1, SACK0, SACK1, SACKe]
   ))
   []
   MEDIUM2 [RACK1, RACK0, SACK1, SACK0, SACKe]
endproc

process RECEIVER [GET, RDT0, RDT1, RDTe, RACK0, RACK1] : noexit :=
      RDT0;        (* correct control bit *)
      GET;         (* message M delivery*)
      RACK0;       (* sending a correct ack *)
      RECEIVER [GET, RDT1, RDT0, RDTe, RACK1, RACK0]
   []
      RDT1;        (* incorrect control bit => *)
      RACK1;       (* sending an incorrect ack *)
      RECEIVER [GET, RDT0, RDT1, RDTe, RACK0, RACK1]
   []
      RDTe;        (* loss indication => *)
      RACK1;       (* sending an incorrect ack *)
      RECEIVER [GET, RDT0, RDT1, RDTe, RACK0, RACK1]
   []
      (hide TIMEOUT in
      TIMEOUT; (* timeout => *)
      RACK1;       (* sending an incorrect ack *)
      RECEIVER [GET, RDT0, RDT1, RDTe, RACK0, RACK1])
endproc
```

Figure 8.16: MEDIUM2 and RECEIVER processes in LOTOS.

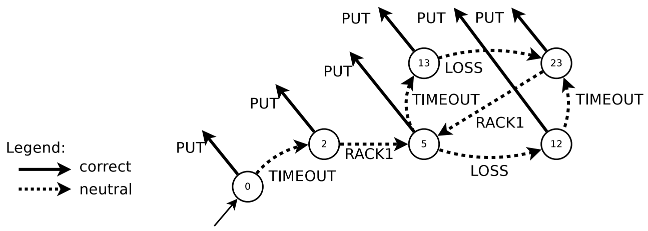


Figure 8.17: Excerpt of the alternating bit protocol LTS model.

Case Study: Multiway Rendezvous Protocol

The last example we describe is the *Multiway Rendezvous Protocol.* This case study represents the evaluation of a formal model, written in LNT, of the multiway rendezvous protocol implemented in DLC (Distributed LNT Compiler) [EL17], a tool that automatically generates a distributed implementation in C of a given LNT specification. The multiway rendezvous protocol must allow processes (called tasks) to synchronize, through message exchange, on a given gate. In this case messages (e.g., abort, commit, ready) are exchanged between two tasks *T1* and *T16* and a gate *A* to synchronize. In Figure 8.18 we show a portion of the LNT specification of the protocol, depicting the parallel construct that synchronizes tasks *T1* and *T16* and a gate *A*. Note that in this case study the gate *A* does not corresponds to an LNT gate construct, but is built through an LNT process.

A synchronization success between *T1* and *T16* on gate *A* is represented by ACTION !DLC_GATE_A !{DLC_TASK_0_T1, DLC_TASK_1_T16}. One of the two tasks cannot execute more than two actions on gate A. This is expressed with the following MCL safety property:

```
[true* . 'ACTION !DLC_GATE_A .*'. true* . 'ACTION !DLC_GATE_A .*'.
true* . 'ACTION !DLC_GATE_A .*'. true*] false
```

We analyse a preliminary faulty version of the protocol which allows performing three synchronizations on gate *A*, that is prohibited by the property (see example *multiway rendezvous* in Table 8.1). We make use of two abstraction techniques to debug this model: the abstracted counterexample, applied to the shortest counterexample, and the version of the shortest path technique where the path must match a given pattern of actions. The first technique shows that the bug is due to a combination of causes, all highlighted by neighbourhoods. We summarise here the main ones. First of all, one of the neighbourhoods (see Figure 8.19a) lets us understand that if a second synchronization on gate *A* is executed instead of the refusal of the negotiation (that is actually executed in the counterexample) the bug does not arise. This means that the refusal of the negotiation by gate *A* is involved in causing the bug. We then detect a set of neighbourhoods, which shows that the bug does not arise if an abort message is received by the task *T16* before the reception of a commit message. Figure 8.19b depicts one example of these neighbourhoods. Finally, the last neighbourhood (see Figure 8.19c) triggers definitively the bad behaviour with the second execution of the synchronization on gate *A*, represented in the form of an incorrect transition.

Our approach shows that the bug comes from a shift between the progress of task *T16* and the rest of the system when a refusal is performed by gate *A*. This is precisely highlighted through the presence in the neighbourhoods of the refusal of the negotiation by gate *A* and the reception of the abort message after the first commit message. Our technique also shows that the causes of the bug are all located before the execution of the second synchronization on gate *A* (included), while the rest of the counterexample is irrelevant from a debugging perspective. This outcome is also confirmed by the second abstraction technique we used.

```
module implem (task_T1, task_T16, data, latest) is

   function gate_A_sync_vect : sync_vect_list is
      return {{ DLC_TASK_0_T1, DLC_TASK_1_T16 }}
   end function

   function global_sync_map : sync_map is
      return {sync_map_entry (dlc_gate_A, gate_A_sync_vect)}
   end function

   process MAIN [ TASK_0_T1_SEND, TASK_0_T1_RECV,
                  TASK_1_T16_SEND, TASK_1_T16_RECV,
                  GATE_A_SEND, GATE_A_RECV,
                  ACTION, HOOK_REFUSE: annonce] is
      par TASK_0_T1_SEND, TASK_0_T1_RECV, TASK_1_T16_SEND, TASK_1_T16_RECV,
          GATE_A_SEND, GATE_A_RECV in
         par
             buffer [TASK_0_T1_SEND, TASK_1_T16_RECV] (DLC_TASK_0_T1, DLC_TASK_1_T16)
          || buffer [TASK_1_T16_SEND, TASK_0_T1_RECV] (DLC_TASK_1_T16, DLC_TASK_0_T1)
          || buffer [TASK_0_T1_SEND, GATE_A_RECV] (DLC_TASK_0_T1, DLC_GATE_A)
          || buffer [GATE_A_SEND, TASK_0_T1_RECV] (DLC_GATE_A, DLC_TASK_0_T1)
          || buffer [TASK_1_T16_SEND, GATE_A_RECV] (DLC_TASK_1_T16, DLC_GATE_A)
          || buffer [GATE_A_SEND, TASK_1_T16_RECV] (DLC_GATE_A, DLC_TASK_1_T16)
         end par
      ||
         par
            MANAGER [TASK_0_T1_SEND, TASK_0_T1_RECV, ACTION]
               (DLC_TASK_0_T1, task_T1_state_space, global_sync_map)
         || MANAGER [TASK_1_T16_SEND, TASK_1_T16_RECV, ACTION]
               (DLC_TASK_1_T16, task_T16_state_space, global_sync_map)
         ||
            GATE [GATE_A_SEND, GATE_A_RECV, ACTION, HOOK_REFUSE]
               (DLC_GATE_A, gate_A_sync_vect)
         end par
      end par
   end process

end module
```

Figure 8.18: Main LNT module of the multiway rendezvous protocol.

We defined two different patterns, one containing a single synchronization and the other one with two synchronizations on gate A. While the abstraction technique used with the single synchronization returned a path to the closest neighbourhood (depicted in a shortened version in Figure 8.20), it did not returned any result using the second pattern, confirming that all neighbourhoods are located before the second synchronization on gate A (included).

Finally, as collateral information, our approach provides the set of labels that are not involved in the counterexample LTS, through the difference between the set of labels of

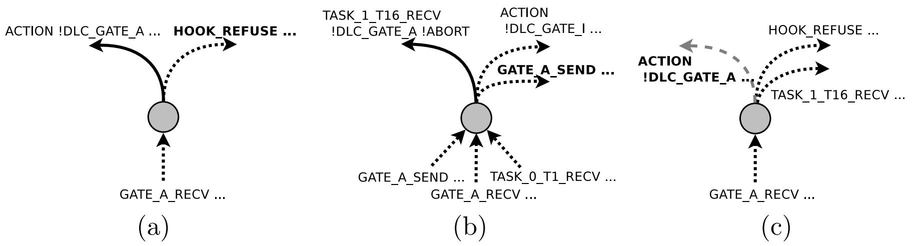


Figure 8.19: Three of the neighbourhoods detected in the abstracted counterexample.

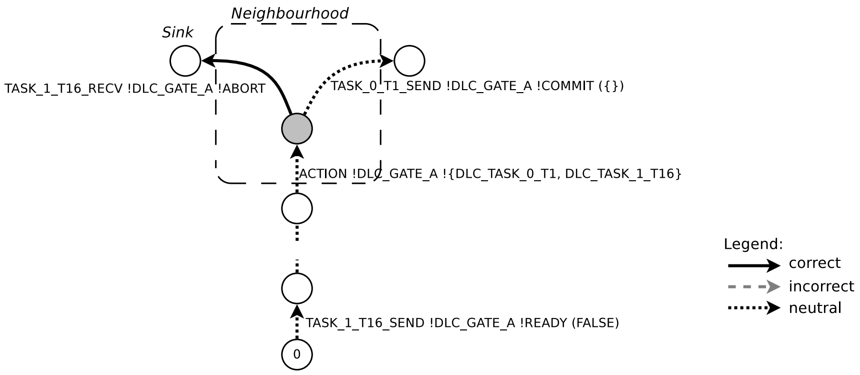

Neighbourhood
Sink
TASK_1_T16_RECV !DLC_GATE_A !ABORT
TASK_0_T1_SEND !DLC_GATE_A !COMMIT ({})
ACTION !DLC_GATE_A !{DLC_TASK_0_T1, DLC_TASK_1_T16}
TASK_1_T16_SEND !DLC_GATE_A !READY (FALSE)
0
Legend:
correct
incorrect
neutral

Figure 8.20: Shortest path to a neighbourhood through a pattern.

the full LTS and the set of labels of the counterexample LTS. Since the counterexample LTS contains all the possible counterexamples, labels that do not belong to it are not involved in the cause of a bug. For instance, in Table 8.1 we see that in this case the number of labels in the counterexample LTS is lower than the one in the full LTS (41 to 53), meaning that about 20% of labels are not related to the bug.

8.2.3 Empirical Evaluation

We conducted an empirical study to validate our approach. We asked 17 developers, with different degrees of expertise, to find bugs on two test cases by taking advantage of the abstracted counterexample techniques. One test case represents a vending machine, while the other one represents a system with three communicating processes (already introduced in Section 8.1). Note that the code provided with both test cases is syntactically correct and compiles. The bug arise from the violation of the given property.

Vending Machine

The system provided with this specification is composed of two processes: a *vending machine* and a *customer*. The vending machine contains 4 bottles of water, 4 bottles of soda and 4 bottles of beer. Money is represented by single unit coins. Each drink has a different cost in terms of coins: 1 coin for the water, 2 for the soda and 3 for the beer. The customer has a wallet containing 4 coins, and she continues buying drinks until she finishes her money or the machine is out of stock. The machine can also provide change if the inserted amount of coins exceeds the cost of the chosen drink and if the customer requires it. The provided property states that if the customer has only 2 coins in his wallet, she should not be able to buy a beer, since it costs 3 coins, and it is written in MCL as follows:

```
[ true*.'CUSTOMER_WALLET !2'.
true*.'DRINK_CHOICE !BEER !+1'.
true*.'PROVIDE_DRINK !BEER'.
true*] false
```

The `CUSTOMER_WALLET !2` label checks the content of the customer's wallet, `DRINK_CHOICE !BEER !+1` expresses the beer choice and `PROVIDE_DRINK !BEER` shows the beer distribution by the machine. This property is not satisfied by the system. This means that, even if the customer only has 2 coins, he is able to buy a beer, which should not be possible. In Figure 8.21 we depict a simplified version of the vending machine process. The bug is in the `else` branch of the `if` statement with the condition `total_coins == 0`. The machine does not update the internal money quantity to 0 when the change is provided.

Concurrent Processes

In this case we reused the specification presented in the Causality Bug example (see Section 8.1.4). We recall here briefly how this specification behaves. The system provided with this specification is composed of three main processes: a producer process, a consumer process and a process that can be both a consumer and a producer. Each process can loop infinitely or break the loop and terminate the execution. A deployer process is also present in order to deploy the three other processes.

The provided property prevents a process to consume if something has not been produced before. The MCL formula states that it should be impossible to have a consumption event (`CONSUME`) which has not been preceded by a production event (`PRODUCE`). This property is written in MCL as follows:

```
[ (not "PRODUCE")* . "CONSUME" . true* ] false
```

This property is not satisfied by the system. In particular, this happens when the `PRODCONS` process (depicted in Figure 8.7 in Section 8.1.4) first chooses to be a consumer, and then does not perform a `SYNC` action.

```
process MACHINE [ PROVIDE_DRINK : DISPENSER_C, DRINK_CHOICE : DRINK_CHOICE_C,
                  COIN_INSERT, RETRIEVE_CHANGE : COINS_C, OP : INTERNAL_OP_C]
                (water_stock, soda_stock, beer_stock : int, rem_change : nat) is
   var  quant, water_q, soda_q, beer_q : int, inserted_coins , tot_coins : nat,
        drink : DRINK_TYPE in
      inserted_coins := 0; tot_coins := rem_change; water_q := water_stock;
      soda_q := soda_stock; beer_q := beer_stock;

      COIN_INSERT(?inserted_coins); tot_coins := inserted_coins + tot_coins;
      DRINK_CHOICE(?drink, ?quant);
      loop L in
         if CHECK_MONEY(drink,tot_coins) then (* Check if inserted money is sufficient. *)
            OP(STOCK_COINS); break L
         end if;
         COIN_INSERT(?inserted_coins); tot_coins := inserted_coins + tot_coins
      end loop

      case drink of DRINK_TYPE in
         WATER ->
            if CHECK_QUANT(water_q, quant) then
               PROVIDE_DRINK(!WATER);
               water_q := water_q - quant;
               tot_coins := tot_coins - DRINK_VALUE(drink)
            end if
         | SODA -> (* similar to WATER case, removed for simplicity *)
         | BEER -> (* similar to WATER case, removed for simplicity *)
      end case;

      if (tot_coins == 0) then
         RETRIEVE_CHANGE(!tot_coins)
      else OP(PROVIDE_COINS); RETRIEVE_CHANGE(!tot_coins);
      end if;
      if ((water_q!=0) or (soda_q!=0) or (beer_q!=0)) then
         MACHINE[PROVIDE_DRINK, DRINK_CHOICE, COIN_INSERT, RETRIEVE_CHANGE, OP]
            (water_q, soda_q, beer_q, tot_coins)
      end if
   end var
end process
```

Figure 8.21: Simplified LNT process of the vending machine.

Evaluation

The developers were divided in two groups, in order to evaluate both specifications with and without the abstracted counterexample. The first group was provided with the vending machine specification without the abstracted counterexample and the communicating processes test case with the abstracted counterexample. We did the opposite with the second group of users. We gave to the users a description of the test case, the LNT specification of the test, the property, a normal counterexample and an abstracted counterexample with an explanation of our method. The developers were asked to discover the bug and measure the total time spent in debugging each specification, then to send us the results by email. When a developer did not discover an actual bug we considered her result as a false positive and we did not take it into account in computing total average times, since it invalidated the final results. Table 8.2 depicts the empirical study results in terms of time, comparing for both test cases the time spent with and without the abstracted counterexample.

	Vending Machine	**Concurrent Sys.**	**Total time**
Classic count.	21.5 min	16 min	19 min
Abstracted count.	20.5 min	10 min	15 min

Table 8.2: Empirical study results.

The total average time spent in finding the bug in both test cases without our techniques is of 19 minutes, while the average time using the abstracted counterexample is of 15 minutes, showing a gain in terms of time with the use of our approach. In general, the gain in time is not that high. This is due to two main reasons. First, the length of the two counterexamples is quite short (27 and 15 actions respectively) w.r.t real cases, making the advantage of our techniques less obvious. This is caused by simple test cases specifications, expressly chosen to allow developers to carry out tests in a reasonable time. The aim of our approach is not to debug simple test cases like these ones, but is rather to complement existing analysis techniques to help the developer when debugging complex real cases. Second, a part of the developers were using our method for the first time, while it is worth noting that our method requires some knowledge in order to use it properly.

We tried to measure also the well known precision and recall values on the empirical study. Precision represents the fraction of relevant results (true positives) among all the retrieved results (true positives + false positives). In our case, it corresponds to the number of true bugs among all bugs discovered by the developers. We computed precision for all tests we carried on, obtaining a total precision value of 0.73%. Recall represents the fraction of relevant results (true positives) among all the relevant elements (true positive + false negatives). However in our empirical study we do not have false nor true negatives, since a bug is always present in each test. Thus, in our case it is not worth computing the recall measure, since the absence of false negatives give maximum recall (100%).

Finally, we also asked developers' opinion about the benefit given by our method in detecting the bug. Over 17 developers, only 4 of them said the abstracted counterexample was not useful or that they did not need it. Almost two-thirds of the developers agreed considering our approach helpful: 8 found that our technique was useful and 3 said that it can be useful in some circumstances (the remaining 2 did not express an opinion).

8.3 Concluding Remarks

Our experiments show that our approach, by exploiting the notion of neighbourhood, can be used to simplify the comprehension of a bug. It is also worth noting that our visualization techniques are not always helpful because, in some cases, nothing can be deduced from the visual model or because the model is too large in terms of number of states/transitions. In those situations, the developer can use one of the abstraction techniques. Finally, is worth stressing that, since our approach applies on the tagged LTS and computes all the neighbourhoods, the returned solution is able to pinpoint all the causes of the property violation, as we have shown precisely in the case studies.

Chapter 9

Conclusion

In this work, we have proposed a novel approach for debugging concurrent systems, in particular for simplifying the comprehension of erroneous behavioural specifications under validation using model checking techniques. To do so, we have introduced the notion of neighbourhoods corresponding to the junction of correct and incorrect transitions in the LTS. Such neighbourhoods represent relevant portions of the LTS which highlight choices between a correct and an incorrect behaviour. Several notions of neighbourhoods have been defined depending on the type of transitions located at such a state (correct, incorrect or neutral transitions). By looking more carefully at those states, we can better understand the source of the bug.

We have proposed two methods for extracting such neighbourhoods. In the first one we have defined a procedure to obtain an LTS containing all the counterexamples given an LTS and a safety property. This LTS containing only counterexamples is then compared to the original LTS for automatically computing all neighbourhoods. The second method focuses on a class of liveness properties, called inevitability properties. We have defined an algorithm to augment the LTS that represents the model of the system with notions of prefixes and suffixes, which express parts of the sequence of inevitable actions. This augmented model is then exploited to compute neighbourhoods.

We have also developed two methods for exploiting neighbourhoods, 3D visualization techniques and abstraction techniques, allowing the developer to first exploit a global visual feedback of the model containing neighbourhoods, and then to go into details and focus on single counterexamples. In this way the developer can obtain precise information that explains the cause of the bug.

We implemented our approach in a tool, called CLEAR, and we evaluated it on several real-world case studies, demonstrating the advantage of both the visualization and the abstraction techniques in practice when adopting the neighbourhood approach. We have also presented several examples of typical bugs with their corresponding visual models. These models exhibit interesting structures that characterize the bug and are helpful for supporting the developer during her debugging tasks. Our experimental study, also supported by an empirical evaluation, shows that our neighbourhood approach helps in practice for more easily pinpointing the source of the bug in the corresponding LTS model.

9.1 Perspectives

As far as future improvements are concerned, we propose here some ideas and research directions to extend our work.

1. Our methods perform a semantic analysis of the LTS model, but we do not take into account the *structure of the specification.* One perspective is to combine the semantic information of the LTS to the syntactic specification from which that model was extracted. This will be achieved by extending our approach with structural information, for instance, in the form of a control flow graph of the specification. The addition of the structural information will allow getting finer results which would help in the identification of the source of the bug. It would also allow the use of code colouring techniques to highlight portions of the original specification, that should be looked at carefully for debugging purposes.

2. In some cases the developer may require information that is not provided by the abstraction techniques detailed in Chapter 4. We thus propose to *extend the set of abstraction techniques* with new ones, based on new notions. For instance, one can define a technique which exploits relations between neighbourhoods. As an example, in the Sanitary Agency case study, presented in Chapter 8, we detected that it was necessary to pass through the first neighbourhood to reach the other two. We could exploit this to extract, given a neighbourhood, all the neighbourhoods that are necessary to traverse in order to reach the former one. A notion of distance to the bug can also be used to build an abstraction technique which retrieves all the neighbourhoods that are at a given distance. For instance, a neighbourhood that is closer to the bug is more relevant than another one that is more distant. One can also classify neighbourhoods according to the probability to reach them from a chosen state (in this case, the neighbourhood with the highest probability should be more interesting), or group them using a similarity notion.

3. It is worth observing that with the counterexample LTS approach the correct part of the LTS is entirely discarded (sink state). However, we can still have some situations in which most of the model is false, resulting in a huge and difficult to visualize LTS. A perspective thus aims at handling the classic state space explosion problem when the model from the specification consists of a large number of states/transitions. One idea in that direction is to rely on clustering techniques (e.g., the ones proposed in [GvH06]) in order to detect repetition of a sub-part of the model (as in the iteration bug in Section 8.1). Once that part of the model is detected, if it does not include

any neighbourhood, it could be collapsed because it is not of interest for finding the bug.

4. The Prefix / Suffix approach has been used for liveness property violations, but we think that it could be applicable also to safety ones. To do this, we propose to search for actions (or sequences of actions) that must not appear, as usually stated by a safety property, instead of searching for sequences of actions that should appear, as we do with the inevitable actions. Consequently in the case of safety properties the correct transitions would be the ones in which the prefix and suffix information is empty. An evaluation with the same case studies we used in this work would also be carried on in order to compare performances of the two approaches.

5. The current *3D visualization* method has been evaluated only with safety property violations, thus taking into account tagged LTSs generated with the Counterexample LTS approach. Our aim is to apply the 3D visualization method to tagged LTSs produced with the Prefix / Suffix approach, thus allowing the use of the methodology proposed in Section 4.3 also with liveness property violations. Visualizing such tagged LTS could result in an increased number of correct transitions, due to the lack of a sink state. However, clustering techniques (e.g., [GvH06]) could be applied in order to reduce the size of correct parts of the tagged LTS.

6. Another perspective consists in building a repository of specifications and visualizations corresponding to *typical bugs*, starting with those presented in Section 8.1. This would allow developers to rely on this repository when trying to understand and correct an erroneous specification by visual analysis. The developers could also contribute by adding their own specifications and visual patterns to the repository.

7. One of the issues we had to cope with during our work was the *lack of erroneous models*. Consequently, we often had to build new faulty models or to introduce faults artificially on existing ones. We thus propose the creation of a test suite of erroneous case studies (in the form of behavioural models, i.e., LTSs), similarly to what have been done with the Siemens Test Suite [HFGO94] in the testing community. Such test suite could be useful also for comparing different debugging techniques.

8. As an ambitious long term perspective, we propose to detect and classify typical bugs in categories (e.g., bugs related to variables, to concurrency, or to non-deterministic choice operators). This could later be useful to build methods to *automatically repair* the bug, thus avoiding the manual intervention of the developer.

Bibliography

[AH90] Hiralal Agrawal and Joseph Robert Horgan. Dynamic program slicing. In *Proc. PLDI'90*, pages 246–256. ACM, 1990.

[AHLW95] Hiralal Agrawal, Joseph R. Horgan, Saul London, and W. Eric Wong. Fault localization using execution slices and dataflow tests. In *Proc. of ISSRE'95*, pages 143–151. IEEE, 1995.

[AKA$^+$02] Kenji Abe, Shinji Kawasoe, Tatsuya Asai, Hiroki Arimura, and Setsuo Arikawa. Optimized substructure discovery for semi-structured data. In *Proc. of PKDD'02*, volume 2431 of *LNCS*, pages 1–14. Springer, 2002.

[ALRL04] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl E. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Sec. Comput.*, 1(1):11–33, 2004.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[BB87] Tommaso Bolognesi and Ed Brinksma. Introduction to the ISO Specification Language LOTOS. *Computer Networks*, 14:25–59, 1987.

[BBC05] A. Bracciali, A. Brogi, and C. Canal. A Formal Approach to Component Adaptation. *Journal of Software Systems*, 74(1), 2005.

[BBC$^+$12] Ilan Beer, Shoham Ben-David, Hana Chockler, Avigail Orni, and Richard J. Trefler. Explaining counterexamples using causality. *Formal Methods in System Design*, 40(1):20–40, 2012.

[BCH$^+$04] Dirk Beyer, Adam Chlipala, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. Generating tests from counterexamples. In *Proc. of ICSE'04*, pages 326–335. IEEE Computer Society, 2004.

[BCR15] Mohammed Bekkouche, Hélène Collavizza, and Michel Rueher. LocFaults: A New Flow-driven and Constraint-based Error Localization Approach. In *Proc. of SAC'15*, pages 1773–1780. ACM, 2015.

[BDM97] Ronald Baecker, Chris DiGiano, and Aaron Marcus. Software visualization for debugging. *Commun. ACM*, 40(4):44–54, 1997.

[BFF09] T. Bultan, C. Ferguson, and X. Fu. A Tool for Choreography Analysis Using Collaboration Diagrams. In *Proc. of ICWS'09*. IEEE, 2009.

[BFT06] Benoit Baudry, Franck Fleurey, and Yves Le Traon. Improving test suites for efficient fault localization. In *Proc. of ICSE'06*, pages 82–91. ACM, 2006.

[BHK$^+$15] Adrian Beer, Stephan Heidinger, Uwe Kühne, Florian Leitner-Fischer, and Stefan Leue. Symbolic Causality Checking Using Bounded Model Checking. In *Proc. of SPIN'15*, volume 9232 of *LNCS*, pages 203–221. Springer, 2015.

[BK08] C. Baier and J.-P. Katoen. *Principles of Model Checking*. MIT Press, 2008.

[BLS17] Gianluca Barbon, Vincent Leroy, and Gwen Salaün. Debugging of Concurrent Systems Using Counterexample Analysis. In *Proc. of FSEN'17*, volume 10522 of *LNCS*, pages 20–34. Springer, 2017.

[BLS18] Gianluca Barbon, Vincent Leroy, and Gwen Salaün. Counterexample Simplification for Liveness Property Violation. In *Proc. of SEFM'18*, volume 10886 of *LNCS*, pages 173–188. Springer, 2018.

[BNR03] Thomas Ball, Mayur Naik, and Sriram K. Rajamani. From symptom to cause: localizing errors in counterexample traces. In *Proc. of POPL'03*, pages 97–105. ACM, 2003.

[BO05] Stefan Blom and Simona Orzan. Distributed state space minimization. *STTT*, 7(3):280–291, 2005.

[BP06] Antonio Brogi and Razvan Popescu. Automated generation of BPEL adapters. In *Proc. of ICSOC'06*, volume 4294 of *LNCS*, pages 27–39. Springer, 2006.

[BR01] Thomas Ball and Sriram K. Rajamani. The SLAM toolkit. In *Proc. of CAV'01*, volume 2102 of *LNCS*, pages 260–264. Springer, 2001.

[BWW14] Mitra Tabaei Befrouei, Chao Wang, and Georg Weissenbacher. Abstraction and Mining of Traces to Explain Concurrency Bugs. In *Proc. of RV'14*, volume 8734 of *LNCS*, pages 162–177. Springer, 2014.

[CAN$^+$01] Jong-Deok Choi, Bowen Alpern, Ton Ngo, Manu Sridharan, and John M. Vlissides. A perturbation-free replay platform for cross-optimized multithreaded applications. In *Proc. of IPDPS'01*, page 23. IEEE Computer Society, 2001.

[CCG$^+$02] Alessandro Cimatti, Edmund M. Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In *Proc. of CAV'02*, volume 2404 of *LNCS*, pages 359–364. Springer, 2002.

[CCG$^+$04] Sagar Chaki, Edmund M. Clarke, Alex Groce, Somesh Jha, and Helmut Veith. Modular verification of software components in C. *IEEE Trans. Software Eng.*, 30(6):388–402, 2004.

[CCG$^+$18] D. Champelovier, X. Clerc, H. Garavel, Y. Guerte, F. Lang, C. McKinty, V. Powazny, W. Serwe, and G. Smeding. Reference Manual of the LNT to LOTOS Translator (Version 6.7). INRIA/VASY and INRIA/CONVECS, 153 pages, 2018.

[CDE$^+$07] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott, editors. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.

[CDLT08] Francesco Calzolai, Rocco De Nicola, Michele Loreti, and Francesco Tiezzi. Tapas: A tool for the analysis of process algebras. *Trans. Petri Nets and Other Models of Concurrency I*, 5100:54–70, 2008.

[CGK$^+$13] Sjoerd Cranen, Jan Friso Groote, Jeroen J. A. Keiren, Frank P. M. Stappers, Erik P. de Vink, Wieger Wesselink, and Tim A. C. Willemse. An overview of the mcrl2 toolset and its recent advances. In *Proc. of TACAS'13*, volume 7795 of *LNCS*, pages 199–213. Springer, 2013.

[CGP01] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model checking*. MIT Press, 2001.

[CGS04] Sagar Chaki, Alex Groce, and Ofer Strichman. Explaining abstract counterexamples. In *Proc. of SIGSOFT FSE'04*, pages 73–82. ACM, 2004.

[CH10] Jürgen Christ and Jochen Hoenicke. Instantiation-based interpolation for quantified formulae. In *Decision Procedures in Software, Hardware and Bioware, 18.04. - 23.04.2010*, volume 10161 of *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany, 2010.

[CJLV02] Edmund M. Clarke, Somesh Jha, Yuan Lu, and Helmut Veith. Tree-Like Counterexamples in Model Checking. In *Proc. of (LICS'02*, pages 19–29. IEEE Computer Society, 2002.

[cle] CLEAR Debugging Tool. https://github.com/gbarbon/clear/.

[CMS$^+$10] J. Cámara, J. Antonio Martín, G. Salaün, C. Canal, and E. Pimentel. Semi-Automatic Specification of Behavioural Service Adaptation Contracts. *Electr. Notes Theor. Comput. Sci.*, 264(1):19–34, 2010.

[CRH10] Hélène Collavizza, Michel Rueher, and Pascal Van Hentenryck. CPBPV: a constraint-programming framework for bounded program verification. *Constraints*, 15(2):238–264, 2010.

[CZ02] Jong-Deok Choi and Andreas Zeller. Isolating failure-inducing thread schedules. In *Proc. of ISSTA'02*, pages 210–220. ACM, 2002.

[CZ05] Holger Cleve and Andreas Zeller. Locating causes of program failures. In *Proc. of ICSE'05*, pages 342–351. ACM, 2005.

[DAC99] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in Property Specifications for Finite-State Verification. In *Proc. of ICSE'99*, pages 411–420. ACM, 1999.

[dSACdLF17] Erickson H. da S. Alves, Lucas C. Cordeiro, and Eddie B. de L. Filho. A method to localize faults in concurrent C programs. *JSS*, 132:336–352, 2017.

[DV90] Rocco De Nicola and Frits W. Vaandrager. Action versus state based logics for transition systems. In *Proc. of Semantics of Systems of Concurrent Processes, 1990*, volume 469 of *LNCS*, pages 407–419. Springer, 1990.

[EBNS13] Tayfun Elmas, Jacob Burnim, George C. Necula, and Koushik Sen. CONCURRIT: a domain specific language for reproducing concurrency bugs. In *Proc. of PLDI'13*, pages 153–164. ACM, 2013.

[EL17] Hugues Evrard and Frédéric Lang. Automatic Distributed Code Generation from Formal Models of Asynchronous Processes Interacting by Multiway Rendezvous. *JLAMP*, 88:33, March 2017.

[ESW12] Evren Ermis, Martin Schäf, and Thomas Wies. Error invariants. In *Proc. of FM'12*, volume 7436 of *LNCS*, pages 187–201. Springer, 2012.

[FBS04] X. Fu, T. Bultan, and J. Su. Conversation Protocols: A Formalism for Specification and Verification of Reactive Electronic Services. *Theoretical Computer Science*, 328(1-2):19–37, 2004.

[FLS06] Giuseppe Di Fatta, Stefan Leue, and Evghenia Stegantova. Discriminative Pattern Mining in Software Fault Detection. In *Proc. of SOQUA'06*, pages 62–69. ACM, 2006.

[FNU03] Eitan Farchi, Yarden Nir, and Shmuel Ur. Concurrent bug patterns and how to test them. In *Proc. of IPDPS'03*, page 286. IEEE Computer Society, 2003.

[GA14] Gregor Goessler and Lacramioara Astefanoaei. Blaming in component-based real-time systems. In *Proc. of EMSOFT'14*, pages 7:1–7:10. ACM, 2014.

[GCKS06] Alex Groce, Sagar Chaki, Daniel Kroening, and Ofer Strichman. Error explanation with distance metrics. *STTT*, 8(3):229–247, 2006.

[GK05] Alex Groce and Daniel Kroening. Making the most of BMC counterexamples. *ENTCS*, 119(2):67–81, 2005.

[GKL04] Alex Groce, Daniel Kroening, and Flavio Lerda. Understanding counterexamples with explain. In *Proc. of CAV'04*, volume 3114 of *LNCS*, pages 453–456. Springer, 2004.

[GL01] Hubert Garavel and Frédéric Lang. SVL: A Scripting Language for Compositional Verification. In *Proc. of FORTE'01*, volume 197 of *IFIP Conference Proceedings*, pages 377–394. Kluwer, 2001.

[GLMS13] Hubert Garavel, Frédéric Lang, Radu Mateescu, and Wendelin Serwe. CADP 2011: A Toolbox for the Construction and Analysis of Distributed Processes. *STTT*, 15(2):89–107, 2013.

[GM13] Gregor Gößler and Daniel Le Métayer. A General Trace-Based Framework of Logical Causality. In *Proc. of FACS'13*, volume 8348 of *LNCS*, pages 157–173. Springer, 2013.

[GM15] Gregor Gößler and Daniel Le Métayer. A general framework for blaming in component-based systems. *Sci. Comput. Program.*, 113:223–235, 2015.

[GMR10] Gregor Gößler, Daniel Le Métayer, and Jean-Baptiste Raclet. Causality analysis in contract violation. In *Proc. of RV'10*, volume 6418 of *LNCS*, pages 270–284. Springer, 2010.

[GMW12] C. Gierds, A. J. Mooij, and K. Wolf. Reducing Adapter Synthesis to Controller Synthesis. *IEEE T. Services Computing*, 5(1), 2012.

[Gro04] Alex Groce. Error explanation with distance metrics. In *Proc. of TACAS'04*, volume 2988 of *LNCS*, pages 108–122. Springer, 2004.

[GS11] G. Gössler and G. Salaün. Realizability of Choreographies for Services Interacting Asynchronously. In *Proc. of FACS'11*, volume 7253 of *LNCS*, pages 151–167. Springer, 2011.

[GS15] Gregor Gößler and Jean-Bernard Stefani. Fault ascription in concurrent systems. In *Proc. of TGC'15*, volume 9533 of *LNCS*, pages 79–94. Springer, 2015.

[GSB07] Andreas Griesmayer, Stefan Staber, and Roderick Bloem. Automated fault localization for C programs. *Electr. Notes Theor. Comput. Sci.*, 174(4):95–111, 2007.

[GSB10] Andreas Griesmayer, Stefan Staber, and Roderick Bloem. Fault localization using a model checker. *Softw. Test., Verif. Reliab.*, 20(2):149–173, 2010.

[GT05] Michael Grottke and Kishor S Trivedi. A classification of software faults. *Journal of Reliability Engineering Association of Japan*, 27(7):425–438, 2005.

[GT09] Hubert Garavel and Damien Thivolle. Verification of GALS systems by combining synchronous languages and process calculi. In *Proc. of SPIN'09*, volume 5578 of *LNCS*, pages 241–260. Springer, 2009.

[GV03] Alex Groce and Willem Visser. What went wrong: Explaining counterexamples. In *Proc. of SPIN'03*, volume 2648 of *LNCS*, pages 121–135. Springer, 2003.

[GvH03] Jan Friso Groote and Frank van Ham. Large State Space Visualization. In *Proc. of TACAS'03*, volume 2619 of *LNCS*, pages 585–590. Springer, 2003.

[GvH06] Jan Friso Groote and Frank van Ham. Interactive visualization of large state spaces. *STTT*, 8(1):77–91, 2006.

[HFGO94] Monica Hutchins, Herbert Foster, Tarak Goradia, and Thomas J. Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proc. of ICSE'94*, pages 191–200. IEEE Computer Society / ACM Press, 1994.

[HG04] Haifeng He and Neelam Gupta. Automated debugging using path-based weakest preconditions. In *Proc. of FASE'04*, volume 2984 of *LNCS*, pages 267–280. Springer, 2004.

[HJMS02] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. In *Proc. of POPL'02*, pages 58–70. ACM, 2002.

[Hol97] Gerard J. Holzmann. The model checker SPIN. *IEEE TSE*, 23(5):279–295, 1997.

[HP05] Joseph Y. Halpern and Judea Pearl. Causes and explanations: A structural-model approach. part i: Causes. *The British Journal for the Philosophy of Science*, 56(4):843–887, 2005.

[igo] Igor tool. https://www.st.cs.uni-saarland.de/askigor/downloads/.

[Inr] Inria CONVECS team. CADP demo 01: Alternating Bit Protocol.

[JHS02] James A. Jones, Mary Jean Harrold, and John T. Stasko. Visualization of test information to assist fault localization. In *Proc. of ICSE'02*, pages 467–477. ACM, 2002.

[JJ93] T. Jéron and C. Jard. Testing for Unboundedness of FIFO Channels. *Theor. Comput. Sci.*, 113(1):93–117, 1993.

[JRS02] HoonSang Jin, Kavita Ravi, and Fabio Somenzi. Fate and Free Will in Error Traces. In *Proc. of TACAS'02*, volume 2280 of *LNCS*, pages 445–459. Springer, 2002.

[Kid98] Peggy Aldrich Kidwell. Stalking the elusive computer bug. *IEEE Annals of the History of Computing*, 20(4):5–9, 1998.

[KL90] Bogdan Korel and Janusz W. Laski. Dynamic slicing of computer programs. *Journal of Systems and Software*, 13(3):187–195, 1990.

[KLM$^+$15] Gijs Kant, Alfons Laarman, Jeroen Meijer, Jaco van de Pol, Stefan Blom, and Tom van Dijk. Ltsmin: High-performance language-independent model checking. In *Proc. of TACAS'15*, volume 9035 of *LNCS*, pages 692–707. Springer, 2015.

[KNP02] Marta Z. Kwiatkowska, Gethin Norman, and David Parker. PRISM: probabilistic symbolic model checker. In *Proc. of TOOLS'02*, volume 2324 of *LNCS*, pages 200–204. Springer, 2002.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[LB12] Stefan Leue and Mitra Tabaei Befrouei. Counterexample explanation by anomaly detection. In *Proc. of SPIN'12*, volume 7385 of *LNCS*, pages 24–42. Springer, 2012.

[LB13] Stefan Leue and Mitra Tabaei Befrouei. Mining Sequential Patterns to Explain Concurrent Counterexamples. In *Proc. of SPIN'13*, volume 7976 of *LNCS*, pages 264–281. Springer, 2013.

[Lew73] David K. Lewis. *Counterfactuals.* Blackwell, 1973.

[Lie97] Henry Lieberman. The debugging scandal and what to do about it (introduction to the special section). *Commun. ACM*, 40(4):26–29, 1997.

[LL13] Florian Leitner-Fischer and Stefan Leue. Causality checking for complex system models. In *Proc. of VMCAI'13*, volume 7737 of *Lecture Notes in Computer Science*, pages 248–267. Springer, 2013.

[LL14] Florian Leitner-Fischer and Stefan Leue. Spincause: a tool for causality checking. In *Proc. of SPIN'14*, pages 117–120. ACM, 2014.

[LM13] Frédéric Lang and Radu Mateescu. Partial Model Checking using Networks of Labelled Transition Systems and Boole an Equation Systems. *Logical Methods in Computer Science*, 9(4), 2013.

[LPSZ08] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *Proc. of ASPLOS'08*, pages 329–339. ACM, 2008.

[LPY97] Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a nutshell. *STTT*, 1(1-2):134–152, 1997.

[LSW08] Stefan Leue, Alin Stefanescu, and Wei Wei. Dependency analysis for control flow cycles in reactive communicating processes. In *Proc. of SPIN'08*, volume 5156 of *LNCS*, pages 176–195. Springer, 2008.

[Mil89] Robin Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[MP11] J. A. Martín and E. Pimentel. Contracts for Security Adaptation. *J. Log. Algebr. Program.*, 80(3-5), 2011.

[MPS08] Radu Mateescu, Pascal Poizat, and Gwen Salaün. Adaptation of service protocols using process algebra and on-the-fly reduction techniques. In *Service-Oriented Computing - ICSOC 2008, 6th International Conference, Sydney, Australia, December 1-5, 2008. Proceedings*, 2008.

[MPS12] Radu Mateescu, Pascal Poizat, and Gwen Salaün. Adaptation of Service Protocols Using Process Algebra and On-the-Fly Reduction Techniques. *IEEE TSE*, 38(4):755–777, 2012.

[MT08] Radu Mateescu and Damien Thivolle. A Model Checking Language for Concurrent Value-Passing Systems. In *Proc. of FM'08*, volume 5014 of *LNCS*, pages 148–164. Springer, 2008.

[OSB14] Meriem Ouederni, Gwen Salaün, and Tevfik Bultan. Compatibility checking for asynchronously communicating software. In *Proc. of FACS'13*, volume 8348 of *LNCS*. Springer, 2014.

[Par81] David Michael Ritchie Park. Concurrency and Automata on Infinite Sequences. In *Proc. of TCS'81*, volume 104 of *LNCS*, pages 167–183. Springer, 1981.

[Pet81] Gary L. Peterson. Myths about the mutual exclusion problem. *Inf. Process. Lett.*, 12(3):115–116, 1981.

[Pol] Polyspace (online). https://www.mathworks.com/products/polyspace.html.

[PT14] Mike Papadakis and Yves Le Traon. Effective Fault Localization via Mutation Analysis: A Selective Mutation Approach. In *Proc. of SAC'14*, pages 1293–1300. ACM, 2014.

[RS04] Kavita Ravi and Fabio Somenzi. Minimal assignments for bounded model checking. In *Proc. of TACAS'04*, volume 2988 of *LNCS*, pages 31–45. Springer, 2004.

[SBR12] G. Salaün, T. Bultan, and N. Roohi. Realizability of Choreographies Using Process Algebra Encodings. *IEEE Transactions on Services Computing*, 5(3):290–304, 2012.

[SBS04] G. Salaün, L. Bordeaux, and M. Schaerf. Describing and Reasoning on Web Services using Process Algebra. In *Proc. of ICWS'04*, pages 43–50. IEEE Computer Society, 2004.

[SEG10] R. Seguel, R. Eshuis, and P. W. P. J. Grefen. Generating Minimal Protocol Adaptors for Loosely Coupled Services. In *Proc. of ICWS'10*. IEEE Computer Society, 2010.

[SEP$^+$13] Gwen Salaün, Xavier Etchevers, Noel De Palma, Fabienne Boyer, and Thierry Coupaye. Verification of a Self-configuration Protocol for Distributed Applications in the Cloud. In *Assurances for Self-Adaptive Systems*, pages 60–79. Springer, 2013.

[SY15] Gwen Salaün and Lina Ye. Debugging Process Algebra Specifications. In *Proc. of VMCAI'15*, volume 8931 of *LNCS*, pages 245–262. Springer, 2015.

[Tar72] Robert Endre Tarjan. Depth-First Search and Linear Graph Algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.

[Tip95] Frank Tip. A survey of program slicing techniques. *J. Prog. Lang.*, 3(3), 1995.

[vdAMSW09] W. M. P. van der Aalst, A. J. Mooij, C. Stahl, and K. Wolf. Service Interaction: Patterns, Formalization, and Analysis. In *Proc. of SFM'09*, volume 5569 of *LNCS*. Springer, 2009.

[VHBP00] Willem Visser, Klaus Havelund, Guillaume P. Brat, and Seungjoon Park. Model checking programs. In *Proc. of ASE'00*, pages 3–12. IEEE Computer Society, 2000.

[WAK$^+$13] Shaohui Wang, Anaheed Ayoub, BaekGyu Kim, Gregor Gößler, Oleg Sokolsky, and Insup Lee. A causality analysis framework for component-based real-time systems. In *Proc. of RV'13*, volume 8174 of *LNCS*, pages 285–303. Springer, 2013.

[Wei79] Mark Weiser. *Program slices: formal, psychological, and practical investigations of an automatic program abstraction method.* PhD thesis, University of Michigan, 1979.

[Wei82] Mark Weiser. Programmers use slices when debugging. *Commun. ACM*, 25(7):446–452, 1982.

[WGG$^+$15] Shaohui Wang, Yoann Geoffroy, Gregor Gößler, Oleg Sokolsky, and Insup Lee. A hybrid approach to causality analysis. In *Proc. of RV'15*, volume 9333 of *LNCS*, pages 250–265. Springer, 2015.

[WGL$^+$16] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. A Survey on Software Fault Localization. *IEEE TSE*, 42(8):707–740, 2016.

[WYIG06] Chao Wang, Zijiang Yang, Franjo Ivancic, and Aarti Gupta. Whodunit? causal analysis for counterexamples. In *Proc. of ATVA'06*, volume 4218 of *LNCS*, pages 82–95. Springer, 2006.

[YHA03] Xifeng Yan, Jiawei Han, and Ramin Afshar. CloSpan: Mining Closed Sequential Patterns in Large Datasets. In *Proc. of SDM'03*, pages 166–177. SIAM, 2003.

[Zak02] Mohammed Javeed Zaki. Efficiently mining frequent trees in a forest. In *Proc. of KDD'02*, pages 71–80. ACM, 2002.

[Zel99] Andreas Zeller. Yesterday, my program worked. today, it does not. why? In *Proc. of ESEC/SIGSOFT FSE'99*, volume 1687 of *LNCS*, pages 253–267. Springer, 1999.

[Zel02] Andreas Zeller. Isolating cause-effect chains from computer programs. In *Proc. of SIGSOFT FSE'02*, pages 1–10. ACM, 2002.

[Zel09] Andreas Zeller. *Why Programs Fail - A Guide to Systematic Debugging, 2nd Edition.* Academic Press, 2009.

[ZH02] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE TSE*, 28(2):183–200, 2002.