Debugging of Behavioural Models using Counterexample Analysis

Gianluca Barbon, Vincent Leroy, and Gwen Salaün

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, and (iii) the counterexample does not explicitly highlight the source of the bug that is hidden in the model. This article presents a new approach that improves the usability of model checking by simplifying the comprehension of counterexamples. To do so, we first extract in the model all the counterexamples. Second, we define an analysis algorithm that identifies actions that make the model skip from incorrect to correct behaviours, making these actions relevant from a debugging perspective. Third, we develop a set of abstraction techniques to extract these actions from counterexamples. Our approach is fully automated by a tool we implemented and was applied on real-world case studies from various application areas for evaluation purposes.

Index Terms—Behavioural Models, Model Checking, Counterexample, Abstraction.

1 Introduction

Recent computing trends promote the development of software applications that are intrinsically parallel, distributed, and concurrent. Designing and developing such systems has always been a tedious and error-prone task, and the ever increasing system complexity is making matters even worse. Although we are still far from proposing techniques and tools avoiding the existence of bugs in a system under development, we know how to automatically chase and find bugs that would be very difficult, if not impossible, to detect manually. The process of finding and resolving bugs is commonly called *debugging*. This 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 this kind of systems, and bugs can be hidden inside parallel behaviours. Thus, there is a need for automatic techniques that can help the developer in detecting and understanding those bugs.

Model checking [1] is an established technique for verifying concurrent systems. It takes as input a model and a property. A model describes all the possible behaviours of a concurrent program and is produced from a specification of the system. In this article, we adopt Labelled Transition Systems (LTS) as model description language. A 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.

Although model checking techniques automatically find bugs in concurrent systems, it is still difficult to interpret the returned counterexamples for several reasons: (i) the counterexample can contain hundreds (even thousands) of actions, (ii) the debugging task is mostly achieved manually (satisfactory automatic debugging techniques do not yet exist), and (iii) the counterexample does not explicitly highlight the source of the bug that is hidden in the model.

This work aims at developing a new approach for simplifying the comprehension of counterexamples and thus favouring usability of model checking techniques. To do this, we propose a method to produce all the counterexamples from a given model and to compare them with the correct behaviours of the model to identify actions that caused the bug. The goal of our approach is to provide techniques for analysing LTS models and assist the user in understanding the cause of the bug by using several defined abstraction techniques.

More precisely, we define a method that 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, we first create an LTS of the formula that represents the property. We then perform the synchronous product between the original LTS and the LTS of the formula. We finally obtain an LTS whose states are simulated by the ones in the original LTS and which contains only traces that violates the property. We call the resulting LTS *counterexample LTS*.

Second, we define an analysis method that identifies actions in the area where counterexamples and correct behaviours, that share a common prefix, split in different paths. We compare the states of the two LTS that belong to this area and we 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

- *G. Barbon and G. Salaün are with the Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG, 38000 Grenoble, France*
- *V. Leroy is with the Univ. Grenoble Alpes, CNRS, Grenoble INP, LIG, 38000 Grenoble, France*

means that they belong to correct paths of the original LTS, which represent behaviours that do not violate the given property. We call these transitions *correct transitions*. States where a correct transition exists are relevant for debugging purposes, since they highlight actions that are responsible for the choice between a correct and a possibly erroneous behaviour. We collect the set of incoming and outgoing transitions of these states and we call this set a *neighbourhood*.

By searching for paths in the counterexample LTS that do not contain any correct transitions we are also able to identify neighbourhoods that correspond to *incorrect transitions*, which consist of transitions that only lead to incorrect behaviours of the LTS. For completeness, we define as *neutral transitions* the transitions which are not correct nor incorrect. The distinction of outgoing transitions allows us to classify neighbourhoods in different types.

We can finally exploit the counterexample LTS, enriched with neighbourhoods, to extract precise information related to the bug, through the use of abstraction techniques. An example of these techniques 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. We also build abstraction techniques that exploit a user input. An example 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.

We have implemented our approach in the CLEAR tool (available online [2]) and validated it on a set of real-world case studies from various application areas. We also build an empirical evaluation in order to test our techniques with real developers. The experiments show that our approach, by exploiting the notion of neighbourhood together with the set of provided abstraction techniques, simplifies the comprehension of the bug.

The work presented in this article is an extension of the paper published in [3]. A precise comparison with our early work is presented in Section 7. The rest of this article is organised as follows. Section 2 introduces LTS models and model checking notions. Section 3 presents the technique for generating the counterexample LTS containing all the counterexamples. Section 4 defines the notion of neighbourhood, presents the process for identifying neighbourhoods in the counterexample LTS and describes the abstraction techniques. In Section 5 we describe our implementation. In Section 6 we apply our tool on real-word examples and we evaluate our approach using an empirical study. Section 7 presents related work while Section 8 concludes this article.

2 PRELIMINARIES

In this work, we adopt *Labelled Transition System (LTS)* as behavioural model of concurrent programs. 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 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. In this work, we use LNT as specification language [4] and compilers from the CADP toolbox [5] for obtaining LTSs from LNT specifications (see Section 5 for more details). However, our approach is generic in the sense that it applies on LTSs produced from any specification language and any compiler/verification tool. 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 $s^0 \xrightarrow{l_1} s_1 \in T, s_1 \xrightarrow{l_2} s_2 \in T, \ldots, s_{n-1} \xrightarrow{l_n} s_n \in T$. 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 characterised 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 $t_\delta(M) = \emptyset$.

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 [6]. In this work, we focus on safety properties, which 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 (we use MCL [7] in Section 6). It can be semantically characterised by an 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) Given an LTS $M = (S, s^0, \Sigma, T)$ and a property φ, a counterexample is any trace which belongs to $t(M) \cap t_\varphi$.

Our solution for counterexample analysis presented in the next section relies on a state matching technique, which takes its foundation into the notion of preorder simulation between two LTSs [8].

Definition 5. (Simulation Relation) Given two LTSs $M_1 = (S_1, s_1^0, \Sigma_1, T_1)$ and $M_2 = (S_2, s_2^0, \Sigma_2, T_2)$, the simulation relation $\sqsubseteq$ between M_1 and M_2 is the largest relation in $S_1 \times S_2$ such that $s_1 \sqsubseteq s_2$ iff $\forall s_1 \xrightarrow{l} s_1' \in T_1$ there exists $s_2 \xrightarrow{l} s_2' \in T_2$ such that $s_1' \sqsubseteq s_2'$. M_1 is simulated by M_2 iff $s_1^0 \sqsubseteq s_2^0$.

3 COUNTEREXAMPLE LTS

In this section, we first introduce the procedure to build an LTS containing all counterexamples (*counterexample LTS*), given a model of the system (*full LTS*) and a temporal property. We then present a technique to generate the matching information between states of the counterexample LTS and states of the full LTS, that we will use in the next section.

3.1 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 6. (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 belong 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 1. 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.

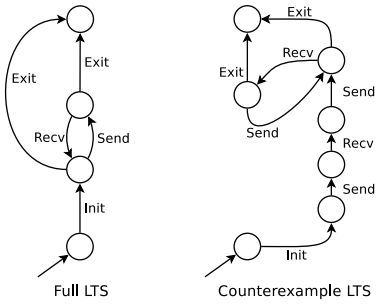

Fig. 1: Simple protocol example: full LTS and counterexample LTS.

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 [9]. 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 apply the subset construction algorithm defined in [10] in order to determinise M_φ. We finally reduce the size of M_φ without changing its behaviour, performing a minimisation based on strong bisimulation [11]. 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 2a. 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.

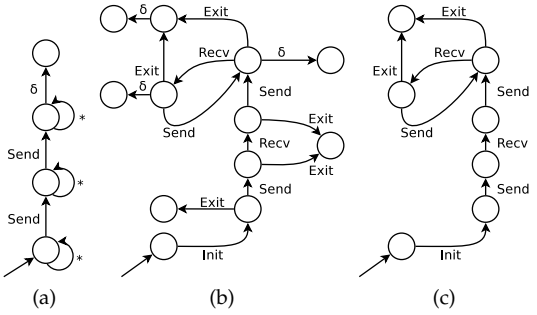

Fig. 2: Simple protocol example: counterexample LTS generation steps.

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 2b shows the result of the product of the full LTS depicted in Figure 1 and the property φ in the form of an LTS depicted in Figure 2a.

Step c) Pruning of the useless transitions generated during the previous step. In particular, we use the pruning algorithm proposed in [12] to remove the traces produced by the synchronous product that are not the prefix of any final trace. As we can see in Figure 2b, 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 2c.

Proposition: *The LTS M_C obtained by this procedure is a counterexample LTS for M_F and φ.*

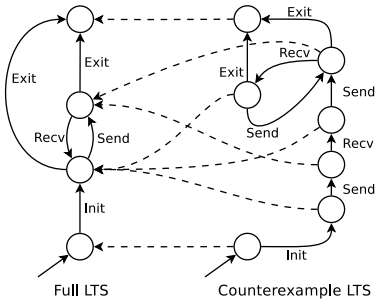

Fig. 3: Simple protocol example: states matching.

3.2 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 Section 2. 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 $x \sqsubseteq y$. 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 3.1. 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 1. 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 3.

4 Counterexample LTS Analysis

In this section we analyse the counterexample LTS produced in the previous section. We first compare the counterexample LTS to the full LTS to identify transitions in locations where traces, that share a common prefix in both LTSs, split in different paths. We call these transitions *correct transitions*. Secondly, we extract the transitions that do not have correct transitions among their successors and we define these transitions as *incorrect transitions*. The correct and incorrect transitions identify relevant portions of the counterexample LTS, described by the notion of *neighbourhood*. Neighbourhoods highlight choices in the model that can lead to behaviours that always satisfy the property, behaviours that always violate the property, or both cases. At the end of the section we present some abstraction techniques, which exploit the notion of neighbourhood to extract precise information from the counterexample LTS to highlight the cause of the bug, and we propose a methodology to use such techniques.

4.1 Neighbourhood Computation

The state matching information, retrieved in Section 3.2, 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 7. (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 LTS. We call these transitions *relevant predecessors*.

Let us illustrate the notion of correct transition on an example. Figure 4 shows a piece of a 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.

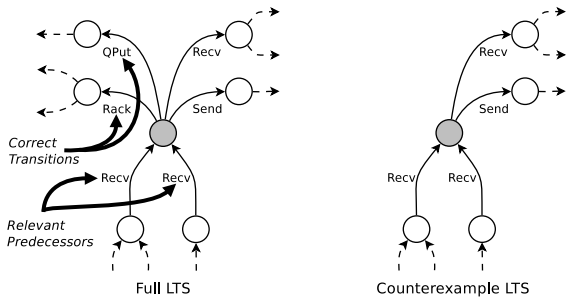

Fig. 4: Example of correct transitions.

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 8. (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 $M_{EC} = (S_{EC}, s_{EC}^0, \Sigma_{EC}, T_{EC})$ where $S_{EC} = S_C \cup s_k$, $s_{EC}^0 = s_C^0$, $\Sigma_{EC} = \Sigma_C \cup \Sigma_{ct}$, and $T_{EC} = T_C \cup T_{ct}$.

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 9. (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 9 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*.

We now add the information concerning the detected transitions (correct, incorrect and neutral transitions) to the LTS in the form of tags. We define the set of transition tags as $\Gamma = \{correct, incorrect, neutral\}$. We represent a tagged transition as $s \xrightarrow{(l,\gamma)} s'$, where $s, s' \in S_{EC}$, $l \in \Sigma_{EC}$ and $\gamma \in \Gamma$. The enriched counterexample LTS where each transition has been tagged is called *tagged LTS*:

Definition 10. (Tagged LTS) Given the enriched counterexample LTS $M_{EC} = (S_{EC}, s^0_{EC}, \Sigma_{EC}, T_{EC})$, obtained from a full LTS M_F and a property φ, and the set of transition tags Γ, the tagged LTS is a tuple $M_T = (S_T, s^0_T, \Sigma_T, T_T)$ where $S_T = S_{EC}$, $s^0_T = s^0_{EC}$, $\Sigma_T = \Sigma_{EC}$, and $T_T \subseteq S_T \times \Sigma_T \times \Gamma \times S_T$.

States where a correct transition exists, or where a sequence of incorrect transitions begins, allow us in a second step to identify a *neighbourhood* in the tagged LTS M_T, which consists of all incoming and outgoing transitions of that state. A neighbourhood represents a choice in the model between two (or more) different behaviours. Behaviours can be correct, incorrect or neutral. Hence, neighbourhoods are branches in the model that directly affect its compliance with the property.

Definition 11. (Neighbourhood) Given the tagged LTS $M_T = (S_T, s^0_T, \Sigma_T, T_T)$, a state $s \in S_T$, such that $\forall t = s' \xrightarrow{(l,\gamma)} s \in T_T$, 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 $t'' = s \xrightarrow{(l,\gamma)} s''' \in T_T$.

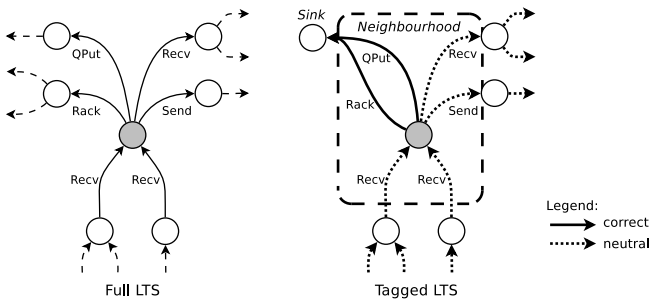

Fig. 5: Example of neighbourhood with correct transitions.

Figure 5 shows the state in the tagged LTS that corresponds to the state in the counterexample LTS depicted in Figure 4. The incoming and outgoing transitions for this state in the tagged LTS correspond to the neighbourhood.

Note that a neighbourhood also contains the so-called relevant predecessor transitions. Relevant predecessors highlight actions performed just before the ones described by the correct (or incorrect) transitions. Since they represent common prefixes for correct and incorrect transitions in neighbourhoods, relevant predecessors are always neutral transitions.

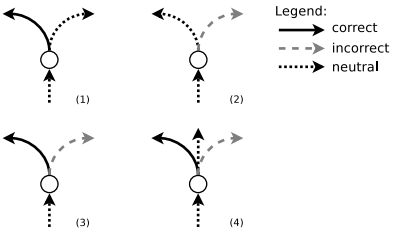

Fig. 6: The four types of neighbourhoods.

4.2 Neighbourhood Taxonomy

We can categorise neighbourhoods in four types by looking at their outgoing transitions (see Figure 6 from left to right and from top to bottom); note that correct transitions are depicted with black lines, incorrect ones are depicted with grey dotted lines and neutral ones are depicted with black dotted lines:

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 7 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*.
4) with at least one *correct transition*, at least one *incorrect transition* and at least one *neutral transition*.

Let us illustrate the neighbourhood taxonomy on the example depicted in Figure 8. The sink state with correct transitions has been added to the counterexample LTS generated in Figure 2. Correct transitions, all representing the *Exit* transition, highlight a neighbourhood in the corresponding source state. In this example, the first two neighbourhoods belong to type 1. The third neighbourhood has both a correct

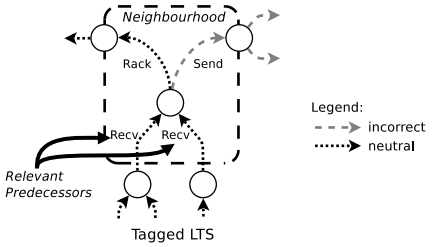

Fig. 7: Example of neighbourhood with an incorrect transition.

and an incorrect transition, corresponding to the second *Send* transition. Consequently, this neighbourhood belongs to type 3.

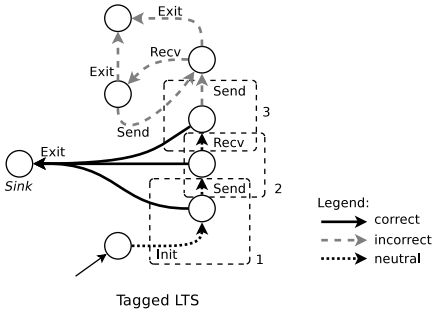

Fig. 8: Simple protocol example: neighbourhood detection.

4.3 Abstraction Techniques

We now define a set of abstraction techniques to provide hints to the developer to discover the source of the bug and thus favour the comprehension of its cause, by exploiting the notion of neighbourhood. We present here in detail two abstraction techniques as well as some alternative versions derived from them. It is worth noting that the abstraction techniques we propose preserve the actions involved in the choices that caused the bug, while they remove actions that are not involved in such choices and thus are less important from a debugging perspective. We will comment on the relevance and benefit of the described abstraction techniques on real-world examples in Section 6.

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, produced from a model M_F 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_F 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.

For illustration purposes, let us consider the counterexample, produced by a model checker from a model M and a property φ, given in the top part of Figure 9. 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 in the bottom part of the figure.

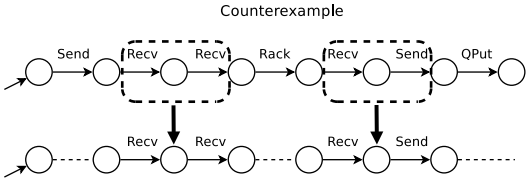

Fig. 9: Example of counterexample abstraction.

Some alternative versions can be derived from this abstraction technique. A first 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. A second alternative 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. This variation of the abstracted counterexample can help to check whether the given pattern is involved in the cause of the bug.

Shortest path to a neighbourhood. The aim of this abstraction technique is to provide a starting point to debug models that contain an 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, produced from a model M_F 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 $t(S_N)$. Let us consider a portion of a tagged LTS M_T depicted in Figure 10. 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 grey show the shortest path from the initial state to the closest neighbourhood.

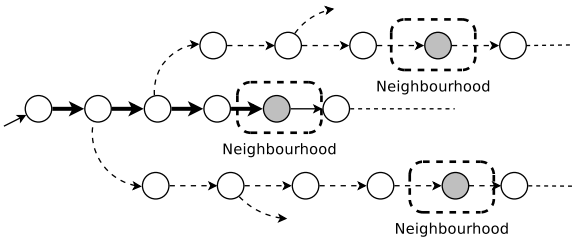

Fig. 10: Example of shortest path between the initial state and a neighbourhood.

Similarly to the abstracted counterexample technique, we developed alternative versions, by refining the searched

type of neighbourhood or by forcing the path to match a pattern of non-contiguous actions.

4.4 Methodology

As far as usability is concerned, here is what we advocate for using our approach from a methodological perspective. First, our method automatically computes all the neighbourhoods in the tagged LTS. In this preliminary step, if the number of detected neighbourhoods is small, the developer can detect particular cases of bugs. For instance, in this step the developer can immediately detect cases in which only a neighbourhood is present and the property is always false.

In a second time, the developer can investigate in detail the bug behaviour by focusing on single counterexamples. To do this, the developer can first use the abstract counterexample technique to obtain a first intuition about the bug cause. Then, she can use the shortest path technique and pattern-based techniques when she needs to refine the results or in particular cases in which the abstracted counterexample technique is not useful.

5 Tool Support

In this section we present the implementation of our approach in the CLEAR tool, which is available online [2]. The CLEAR tool architecture is depicted in Figure 11 and consists of two main modules: the neighbourhood calculation module and the analysis module. The former one has been implemented in Java (about 3,500 lines of code) and is divided into two components. The first component implements the counterexample LTS generation step described in Section 3.1. It relies on the CADP toolbox [5], which enables one to specify and analyse concurrent systems using model and equivalence checking techniques. The computation of the counterexample LTS is achieved by a script we wrote using SVL [13], a scripting language that allows one to interface with tools provided in the CADP toolbox. This script takes as input a specification written in the LNT [4] process algebra and an MCL [7] property. After the generation of the counterexample LTS (in the form of AUT file) through the SVL script, the tool takes as input such model and stores it in memory using a Java graph modelling library. 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 later used to discover the neutral and incorrect ones in the enriched counterexample LTS, thus producing the tagged LTS.

Neighbourhoods are detected in the second component 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 tagged LTS.

Finally, the analysis module provides the implementation of the abstraction techniques described in Section 4.3. We use the Neo4j [14] graph database to store the tagged LTS

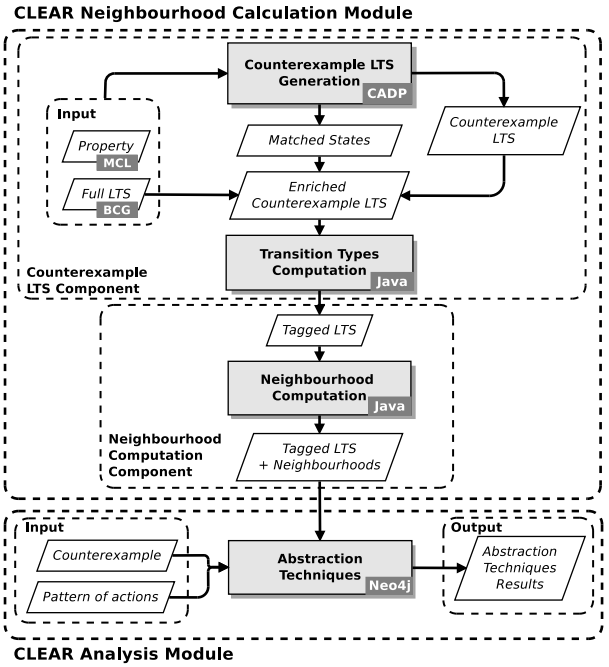

Fig. 11: Overview of the tool support.

and we built abstraction techniques as Neo4j queries. 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. 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 made in Section 4.2). Queries are built using the Cypher language, a graph query language developed for the Neo4j graph database. For instance, the shortest path from the initial node to a neighbourhood is retrieved with the following Cypher query:

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

Visual rendering of the tagged LTS is also provided through the Neo4j GUI.

6 Evaluation

In this section, we apply our approach to several case studies found in the literature and we describe an empirical study we built to evaluate our approach with some developers. In the evaluation of our experimental results we address the following research questions:

RQ1: *Is our approach able to reduce the counterexample size?*
RQ2: *Does our approach provide an automatic debugging method which makes easier the work usually performed by the developer?*
RQ3: *Can our approach highlight all the sources of the bug that are hidden in the model?*

At the end of this section we will answer to these questions and we will discuss about potential threats to validity of our evaluation results. The three case studies detailed

in Section 6.1 and the empirical case studies described in Section 6.2 are available online [2].

6.1 Experimental Results

We carried out experiments on about 100 real-world examples. For each one, we use as input an LNT specification (following the original specification) or an LTS model, and a safety property. Table 1 summarises the results for some of these experiments. The first column contains the name of the model. 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 following columns give the number of neighbourhoods for each neighbourhood type (*cn*: with only correct transitions, *in*: with only incorrect transitions, *ci*: with correct and incorrect transitions, *cni*: with correct, incorrect and neutral transitions), and the percentage of neighbourhoods over the number of states in the tagged LTS. We also present in the table the results of the counterexample abstraction technique, in terms of size of the shortest 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 computation and the neighbourhood computation.

First of all, as far as computation time is concerned, the table shows 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 Java part of our tool. 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 hundred of states).

Second, 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 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 (line 9 in Table 1), mean that we identified only one neighbourhood in the shortest counterexample.

There exist some particular cases in which the abstracted counterexample contains only one action. 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 (lines 19-20 in Table 1), in which we have introduced a bug, and that we have verified with a property that guarantees mutual exclusion. The tagged LTS with the initial neighbourhood allows us 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 executions (thus no counterexample is produced), the counterexample LTS is not generated and no neighbourhood is found (see the *restaurant booking* case at line 21 in Table 1).

Finally, note that in the case of the TFTP/UDP protocol (lines 33-36 in Table 1) , 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 tagged LTS contains a final transition, and it is this final transition itself which is incorrect. In this specific case where the abstracted counterexample technique is not helpful, the developer can use other techniques (for instance the shortest path to a neighbourhood) to retrieve useful information about the bug source.

We now present three case studies from Table 1.

6.1.1 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 to defeat 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 12 shows the LNT process for a player. LNT processes are built from correct termination (**null**), actions, sequential composition (**;**), conditional construct (**if**), assignment (**:=**), nondeterministic choice (**select**), parallel composition (**par**), and looping behaviour (**loop**). The reader interested in more details about LNT should refer to [4]. We discuss here one of the shifumi tournaments case studies described in Table 1 (line 16), which represents a tournament between three players. A property is provided to guarantee no cheating by any player. More precisely, it avoids that a player that has previously lost can play again in the tournament.

```
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
```

Fig. 12: Shifumi tournament: LNT code for a Shifumi player.

The analysed tournament contains a bug, since a dishonest player is able to play again after having lost a game. We have generated the tagged 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

Example	L_F (s/t/l)	L_C (s/t/l)	$cn/in/ci/cin$	$nb.\%$	$\|Ce\|$	$\|Ce_r\|$	$time$
1. sanitary agency (v.1)	227 / 492 / 31	226 / 485 / 31	6 / 10 / 0 / 0	7.08 %	14	2	7.6s
2. sanitary agency (v.2)	142 / 291 / 31	526 / 1064 / 31	12 / 10 / 4 / 2	5.32 %	64	7	7.7s
3. sanitary agency (v.3)	91 / 172 / 31	55 / 95 / 23	5 / 5 / 2 / 0	21.82 %	19	6	7.9s
4. SSH protocol (v.1)	23 / 25 / 23	24 / 24 / 19	1 / 0 / 1 / 0	8.33 %	14	3	7.6s
5. SSH protocol (v.2)	23 / 25 / 23	40 / 40 / 19	3 / 0 / 1 / 0	10.00 %	30	7	7.7s
6. client supplier (v.1)	35 / 45 / 26	29 / 33 / 24	2 / 0 / 1 / 0	10.34 %	18	5	7.6s
7. client supplier (v.2)	35 / 45 / 26	25 / 25 / 24	3 / 0 / 1 / 0	16.00 %	19	6	7.7s
8. client supplier (v.3)	35 / 46 / 26	33 / 41 / 24	1 / 2 / 1 / 0	12.12 %	16	4	7.9s
9. train station	39 / 66 / 18	26 / 34 / 18	0 / 2 / 1 / 0	11.54 %	6	2	7.9s
10. selfconfig	314 / 810 / 27	159 / 355 / 27	24 / 15 / 1 / 5	28.30 %	14	2	7.8s
11. CFSM	1321 / 2563 / 7	3655 / 7246 / 7	12 / 205 / 2 / 0	5.99 %	7	2	7.8s
12. online stock broker	1331 / 2770 / 13	3516 / 7326 / 13	44 / 145 / 17 / 0	5.86 %	23	2	7.9s
13. multiway rendezvous (simple)	2171 / 5098 / 53	171 / 283 / 36	89 / 2 / 1 / 1	54.39 %	47	38	8.2s
14. multiway rendezvous	1318 / 3217 / 53	539 / 1186 / 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 / 1814 / 67	30 / 53 / 10 / 18	22.24 %	7	2	7.9s
17. shifumi (4 players)	1362 / 7209 / 125	3577 / 19233 / 125	206 / 418 / 15 / 188	23.12 %	7	2	10.6s
18. shifumi (5 players)	9693 / 70596 / 201	25593 / 188466 / 201	1622 / 2999 / 20 / 1598	24.38 %	7	2	12.2s
19. Peterson algorithm (v.1)	352112 / 552848 / 85	673569 / 1058113 / 85	0 / 1 / 0 / 0	0.00 %	35	1	24.2s
20. Peterson algorithm (v.2)	352112 / 552848 / 85	683958 / 1074948 / 85	0 / 1 / 0 / 0	0.00 %	35	1	24.1s
21. restaurant booking	55 / 78 / 31	-	0 / 0 / 0 / 0	0.00 %	0	0	7.2s
22. travel agency	60 / 99 / 26	60 / 99 / 26	0 / 1 / 0 / 0	1.67 %	22	1	7.6s
23. FTP transfer	49 / 86 / 18	45 / 76 / 18	0 / 7 / 1 / 2	22.22 %	10	2	7.6s
24. news server	21 / 34 / 8	14 / 18 / 6	0 / 1 / 1 / 2	28.57 %	4	2	7.7s
25. mars explorer	52 / 74 / 34	45 / 66 / 26	0 / 0 / 1 / 0	2.22 %	23	2	7.6s
26. factory job manager	30 / 45 / 14	18 / 21 / 14	2 / 0 / 1 / 0	16.67 %	6	4	7.9s
27. vending machine	17 / 19 / 16	13 / 13 / 12	0 / 0 / 1 / 1	15.38 %	9	3	7.5s
28. restaurant service	25 / 37 / 12	23 / 33 / 12	0 / 1 / 1 / 0	8.70 %	9	2	7.4s
29. CFSM (estelle)	4058 / 8219 / 9	24171 / 50830 / 9	1344 / 973 / 16 / 9	9.69 %	18	2	8.5s
30. reactive system	266 / 720 / 5	296 / 786 / 5	0 / 1 / 0 / 0	0.34 %	3	1	7.8s
31. bug repository report	80 / 158 / 13	108 / 203 / 13	0 / 1 / 0 / 0	0.93 %	15	1	7.6s
32. message exchange	666 / 2281 / 20	1166 / 3857 / 20	0 / 74 / 0 / 74	12.69 %	17	2	7.9s
33. TFTP/UDP protocol (v.1)	4 / 7 / 2	5 / 11 / 2	0 / 1 / 0 / 0	20.00 %	1	1	7.7s
34. TFTP/UDP protocol (v.2)	98205 / 9018043 / 11	214217 / 19852120 / 11	2957 / 2444 / 1 / 16	2.53 %	3	3	558.3s
35. TFTP/UDP protocol (v.3)	61008 / 6328658 / 11	123983 / 12318353 / 11	6323 / 5758 / 0 / 1556	11.00 %	8	8	301.5s
36. TFTP/UDP protocol (v.4)	98205 / 9018043 / 11	214217 / 19852616 / 11	2955 / 2444 / 1 / 16	2.53 %	3	3	571.5s

TABLE 1: Experimental results.

involves 5 neighbourhoods in the tagged LTS. The detected neighbourhoods precisely identify how the bug originates. Indeed they allow us to understand that player one is the cheater and that she 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 (Figure 13a). 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 (Figure 13b). 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. 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 (Figure 13c).

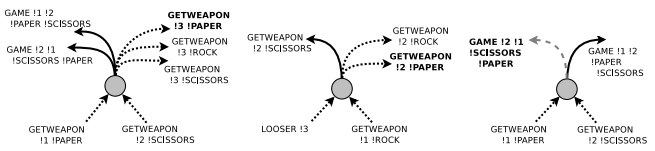

Fig. 13: Shifumi tournament: three of the neighbourhoods detected in the abstracted counterexample.

The shortest path to a neighbourhood technique (depicted in Figure 14) 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

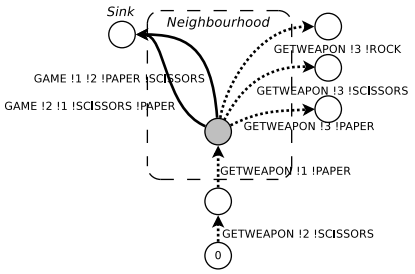

Fig. 14: Shifumi tournament: shortest path to a neighbourhood.

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.

6.1.2 Case Study: Sanitary Agency

We now describe the *sanitary agency* [15] example (line 3 in Table 1), which models an agency that aims at supporting elderly citizens in receiving sanitary assistance from the public administration. A bank model is defined to manage fees and payments, while a cooperative model is built to provide transportation and meal services. A citizen and

a sanitary agency models are also defined. The four LTS models are depicted in Figure 15.

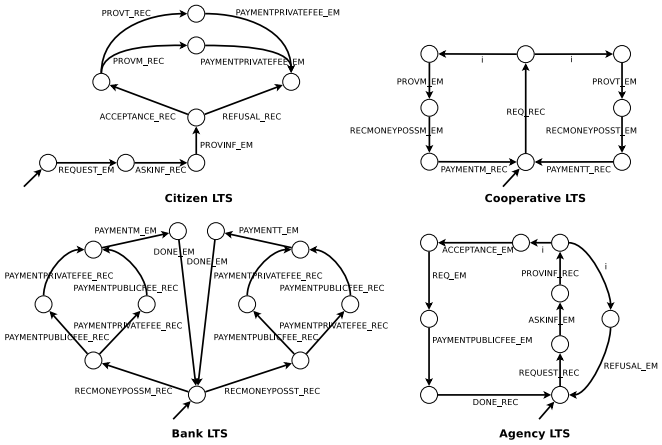

Fig. 15: Sanitary agency: LTS models.

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 tagged LTS, divided into five neighbourhoods from correct transitions, five from incorrect transitions and two from correct and incorrect transitions (without neutral transitions).

We applied the abstracted counterexample technique to the shortest counterexample. The abstracted counterexample involves four neighbourhoods, and this allows us to reduce its size from 19 actions to only 6 actions. Figure 16 shows (from left to right) the full LTS of the sanitary agency model, the abstracted 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 abstracted counterexample. The first cause of violation is emphasised 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 which 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. For instance, the first neighbourhood is of the first type, highlighting a choice that leads to correct behaviour, while the third neighbourhood is of the second type, thus highlights a choice that leads to incorrect behaviour. Finally, note that in this specific case the abstracted counterexample technique was sufficient to understand the cause of the bug. The shortest path to a neighbourhood technique would not be useful, since the first neighbourhood identified with the abstracted counterexample is common to all the counterexamples.

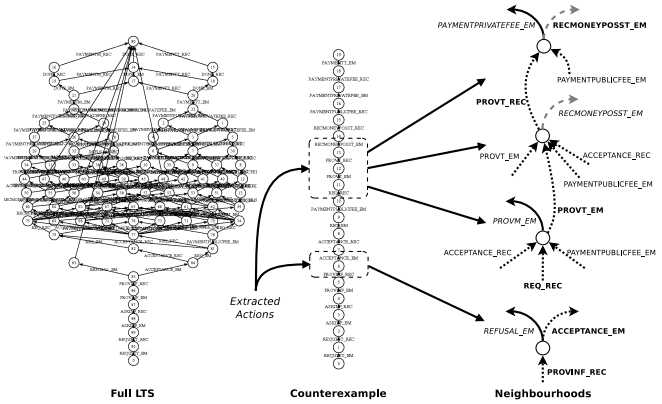

Fig. 16: Sanitary agency: full LTS and abstracted counterexample.

6.1.3 Case Study: Multiway Rendezvous Protocol

The last example we present 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) [16], 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. Note that in this case study the gate *A* does not correspond 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 property:

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

We have analysed a preliminary faulty version of the protocol (line 14 in Table 1) which allows performing three synchronizations on gate *A*, that is prohibited by the property. We made use of two abstraction techniques to debug this model: the abstracted counterexample, applied to the shortest counterexample, and the version of the shortest path to a neighbourhood 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 (Figure 17a) 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 17b depicts one example of these neighbourhoods. Finally, the last neighbourhood (Figure 17c) triggers definitively the bad behaviour with the second execution of the synchronization on gate *A*, represented in the form of an incorrect transition.

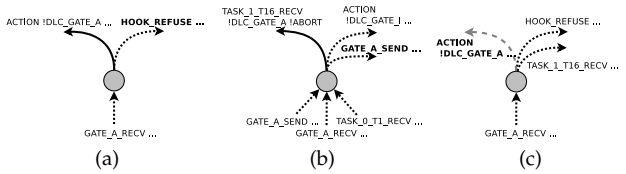

Fig. 17: Multiway rendezvous protocol: three of the neighbourhoods detected in the abstracted counterexample.

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. 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, 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 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 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.

6.2 Empirical Study

We built an empirical study to validate our approach. We asked to 17 developers, with different degrees of expertise, to find bugs on two test cases by taking advantage of the abstracted counterexample technique. The two test cases were specifically developed for the empirical study and modified to introduce a bug. The first one represents a vending machine, in which a bug in the change allows the customer to buy drinks even if there is not enough money. The second test case is a system with three communicating processes (a producer, a consumer and a process that can be both), in which a bug in the synchronization allows a process to consume even if nothing has been produced yet.

The developers were divided in two groups, in order to evaluate both test cases with a classic counterexample and with the abstracted counterexample. The first group was provided with the vending machine test case with a classic counterexample and the communicating processes test case with the abstracted counterexample. We did the opposite for the second group of users. We gave to the users a description of the test cases, the LNT specifications of the tests, the properties, the classic counterexamples and an abstracted counterexample with an explanation of our method (only in one of the two tests). The developers were asked to discover the bug and measure the total time spent in debugging each test case, then to send us the results by email. 70% of the developers found the correct bug with the classic counterexample, while 77% found the correct bug with the abstracted counterexample. When a developer did not discover an actual bug in one of the two tests we considered this result as a false positive and we did not take it into account in computing total average times. Note that, if the other test was correct, we took it into account.

	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 2: Empirical study: average time results.

Table 2 depicts the empirical study results in terms of time, comparing for both test cases the time spent with the classic counterexample and with the abstracted one. 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. Figure 18 depicts the time distributions charts for both test cases, with the classic counterexample and with the abstracted one. Missing columns represent cases in which the developer did not discover an actual bug.

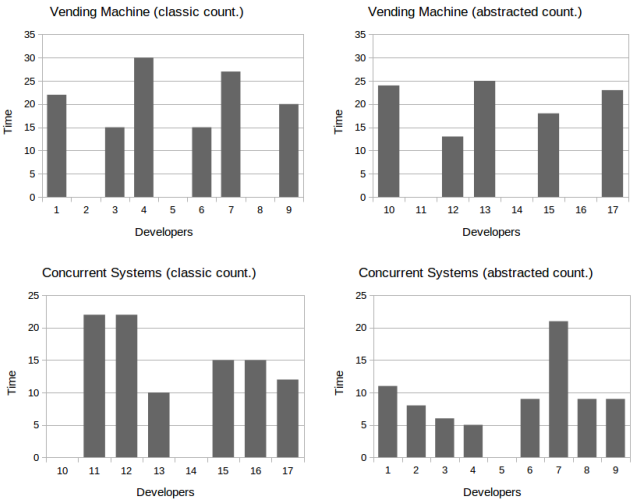

Fig. 18: Empirical study: time distributions.

We also computed the median time spent in both test cases. Median time for vending machine with the classic counterexample is of 21 minutes, while it is slightly higher

with the abstracted one (23 minutes). On the contrary in the concurrent system test case, median values are of 15 minutes with the classic counterexample, and of 9 minutes with the abstracted one, showing a time improvement. It is worth noting that the results we obtained in the concurrent system case are better than in the vending machine one. This is because the bug on the vending machine is caused by a missing value update, requiring the developer to reason about variables accesses, which are not easy to spot for the developer when she has not been involved in the implementation of the specification.

We also asked developers' opinion about the benefit given by our method in simplifying the detection of the bug. Over 17 developers, only 23% of them said the abstracted counterexample was not useful or that they did not need it. 65% of the developers agreed considering our approach helpful: 47% of them found that our technique was useful and 18% said that it can be useful in some circumstances (the remaining 12% did not express an opinion).

Finally, given the small sample size, one may wonder whether the improvements obtained with the use of the abstracted counterexample technique could have been caused by random chance. Therefore, we have carried out a statistical analysis of the empirical study results. More precisely, we have computed the 95% confidence interval of the time difference and the p-value using a two-tailed test. As far as the Vending Machine case is concerned, the interval is [-6.89,8.69] and the p-value is 0.7996, which is not statistically significant. However in the Concurrent System case the interval is [0.41,12.09], and the p-value is 0.0379, which is generally accepted as statistically significant ($p \leq 0.05$).

6.3 Results Discussion

Experiments presented in Table 1 with the abstracted counterexample technique show a clear gain in length w.r.t the original counterexample, allowing us to answer to research question **RQ1**. Our case studies show that our approach, with the help of the chosen abstraction technique, allows the developer to identify the cause of the property violation by identifying specific actions and removing noise in counterexamples thanks to the notion of neighbourhood. Our empirical study shows that the automatic computation of the neighbourhoods with the use of the chosen abstraction technique makes the debugging approach easier. This is also confirmed by the developer satisfaction percentages we obtain, thus answering our research question **RQ2**. Finally, since our approach applies on the tagged LTS and computes all the neighbourhoods, the returned solution is able to pinpoint all the sources of the property violation, as we have shown precisely in the case studies, thus allowing us to answer to **RQ3**.

6.4 Threats to Validity

We discuss here some threats to validity of the results of our evaluation work. First of all, one of the issues we had to cope with during our work was the lack of erroneous models. A benchmark of erroneous case studies (in the form of behavioural models), similar to the Siemens Test Suite [17] for the testing community, does not exist yet. Consequently, either we reused existing real-world models and artificially introduced bugs, either we had to build new faulty models.

For what concerns the empirical study, the average gain in time is not that high. This is due to three main reasons. First, the length of the two counterexamples is quite short (27 and 15 actions respectively) w.r.t. real-world cases, making the advantage of our techniques less obvious. This is caused by simple 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 specifications, which can take hours to debug. Second, a part of the developers were using our method for the first time, while our method requires some knowledge in order to use it properly. The gain in time brought by our approach w.r.t. a classic counterexample might thus not be visible for such developers. Third, the number of developers involved in the empirical study is rather low (only 17), thus potentially weakening the results pertinence.

7 Related Work

In this section, we review existing results for debugging hardware/software systems with a particular focus on causality analysis, counterexample explanation approaches, and testing-based fault localization.

Causality analysis. Causality analysis aims at relating causes and effects, which can help in debugging faults in systems. This analysis relies on a notion of counterfactual reasoning, where alternative executions of the system are derived by assuming changes in the program. There have been several papers published on this subject recently, see, *e.g.* [18], [19], [20], [21]. Let us mention a couple of them with more details. In [20], the authors present a general approach for causality analysis of system failures based on component specifications and observed component traces. This approach uses counterfactual reasoning and blames components based on a single execution trace violating a safety property. In [21], the authors choose the Halpern and Pearl model to define causality checking. In particular, they analyse traces of counterexamples generated by bounded model checking to localise errors in hardware systems. We observed that our approach was useful for detecting bugs that could have been detected using causality analysis techniques, such as [20] and [21]. This is the case of the neighbourhoods detected with the abstracted counterexample technique in the Sanitary Agency case study (Section 6.1.2), where the bug had a first cause in the choice of the first neighbourhood. However, this was not our focus when developing our solution, which was on the analysis of choices made in the specification and their impact on the validity of a given property.

Counterexample explanation. Another line of research focuses on interpreting counterexample and favouring their comprehension, see, *e.g.* [22], [23], [24], [25], [26], [27]. Let us introduce with more details two recent works which perform counterexample analysis by applying pattern mining techniques. In [27], 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 [28]. CloSpan is also adopted in [26], where the authors apply sequential pattern mining to traces of counterexamples to reveal unforeseen interleavings that may be a source of error. However, reasoning on traces as achieved in [26], [27] induces several issues. 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. It is also worth noting that the approaches presented in [26], [27] are usually more scalable for large systems and do not require a complete model of the system, which may be difficult to obtain for real software artefacts.

Two other works have a specific focus on a finer analysis of counterexample traces that is more similar to our approach. In [22] the authors propose a method to interpret counterexamples traces from liveness properties by dividing them into fated and free segments. Fated segments represents 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 [22] 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. Moreover our approach has a specific focus on safety properties, while they focus on liveness properties.

In [24] the authors propose automated methods for the analysis of 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). By relying on a notion of control location, their method tries to make sure that such variations are for one bug to avoid multi-bug confusions. The authors then propose various methods to extract common features and differences between the two sets in order to provide feedbacks to the user. Three different extraction methods are proposed: transition analysis, invariant analysis and transformation of positives into negatives. Similarly to our work, the work in [24] also wants to better explain the counterexample with a focus on safety properties. However, while our approach has a global view on the whole LTS and focuses on the comparison of the full and the counterexample LTS to locate choices that affect the property satisfaction, their method relies on the analysis of a single counterexample and its variations, making sure that negative variations are from the same bug.

Fault localization using testing. 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 method, statistical methods, or machine learning approaches, see [29] for a survey. We focus here on fault localization using testing approaches, see, *e.g.* [30], [31], [32], which is quite related to our approach since testing can be seen as a case of non-exhaustive model checking. Let us introduce as an example a recent approach proposed by Le Traon and Papadakis using a mutation-based fault localization approach [32]. This paper suggests the use of a sufficient mutant set to locate effectively the faulty statements. Experiments carried out in this paper reveal that mutation-based fault localization is significantly more effective than current state-of-the-art fault localization techniques. Note that this mutation analysis approach applies on sequential C programs under validation using testing techniques whereas we focus here on formal models of concurrent programs being analysed using model checking techniques.

Previous work. We published in [3] a preliminary version of our approach for counterexample analysis of safety property violations, which has been extended in this work from multiple perspectives. First of all, in the current work we introduced the detection of new types of transitions (neutral and incorrect), that allow us to improve the notion of neighbourhood and to recognise different neighbourhood types. Neighbourhood types favour a finer categorization of choices. Indeed, they allow to detect also choices that lead only to incorrect behaviours, while in [3] we had a focus on the neighbourhood generated by correct transitions. Secondly, we expanded the number of techniques that the developer can use for debugging the model. Indeed, the abstracted counterexample now represents only one of the abstraction techniques that can be chosen by the developer depending on the required information, and which allow a more complete analysis of the model. Finally, we improved the experimental evaluation of our approach by: (i) revising the results of test cases presented in [3] by taking into account the new transitions types; (ii) extending the set of test cases with new ones, by considering also models with an high number of transitions; (iii) building an empirical evaluation with real developers, which confirmed the benefits of our approach.

8 Conclusion

In this article, we have proposed a novel method for debugging behavioural models based on the analysis of counterexamples produced by model checking techniques. First, we have defined a procedure to obtain an LTS containing all the counterexamples given a full LTS and a safety property. Second, we have introduced the notion of neighbourhoods corresponding to the junction of correct and incorrect transitions in the LTS. We have defined several notions of neighbourhoods depending on the type of transitions located at such a state (correct, incorrect or neutral transitions). We have also proposed an algorithm for automatically computing all neighbourhoods by comparing the full LTS and the LTS consisting of all counterexamples. Third, we have proposed several abstraction techniques to exploit the previously computed neighbourhoods for providing abstracted counterexamples useful for debugging purposes. Finally, we

have implemented our approach as a tool and evaluated it on several real-world case studies. 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.

ACKNOWLEDGMENTS

We are grateful to Frédéric Lang and Radu Mateescu for their valuable suggestions to improve the article. We would like to thank Hugues Evrard as well for his help in providing some of the test cases.

REFERENCES

[1] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model checking*. MIT Press, 2001.
[2] "CLEAR Debugging Tool." 2019, https://github.com/gbarbon/clear/.
[3] G. Barbon, V. Leroy, and G. Salaün, "Debugging of Concurrent Systems Using Counterexample Analysis," in *Proc. of FSEN'17*, ser. LNCS, vol. 10522. Springer, 2017, pp. 20–34.
[4] 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)," 2018, INRIA/VASY and INRIA/CONVECS.
[5] H. Garavel, F. Lang, R. Mateescu, and W. Serwe, "CADP 2011: A Toolbox for the Construction and Analysis of Distributed Processes," *STTT*, vol. 15, no. 2, pp. 89–107, 2013.
[6] C. Baier and J. Katoen, *Principles of Model Checking*. MIT Press, 2008.
[7] R. Mateescu and D. Thivolle, "A Model Checking Language for Concurrent Value-Passing Systems," in *Proc. of FM'08*, ser. LNCS, vol. 5014. Springer, 2008.
[8] D. M. R. Park, "Concurrency and Automata on Infinite Sequences," in *Proc. of TCS'81*, ser. LNCS, vol. 104. Springer, 1981, pp. 167–183.
[9] F. Lang and R. Mateescu, "Partial Model Checking using Networks of Labelled Transition Systems and Boole an Equation Systems," *Logical Methods in Computer Science*, vol. 9, no. 4, 2013.
[10] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.
[11] R. Milner, *Communication and Concurrency*. Prentice Hall, 1989.
[12] R. Mateescu, P. Poizat, and G. Salaün, "Adaptation of Service Protocols Using Process Algebra and On-the-Fly Reduction Techniques," *IEEE TSE*, vol. 38, no. 4, pp. 755–777, 2012.
[13] H. Garavel and F. Lang, "SVL: A Scripting Language for Compositional Verification," in *Proc. of FORTE'01*, ser. IFIP Conference Proceedings, vol. 197. Kluwer, 2001.
[14] "Neo4j Graph Database." 2019, https://neo4j.com/.
[15] G. Salaün, L. Bordeaux, and M. Schaerf, "Describing and Reasoning on Web Services using Process Algebra," in *Proc. of ICWS'04*. IEEE Computer Society, 2004, pp. 43–50.
[16] H. Evrard and F. Lang, "Automatic Distributed Code Generation from Formal Models of Asynchronous Processes Interacting by Multiway Rendezvous," *JLAMP*, vol. 88, p. 33, Mar. 2017.
[17] M. Hutchins, H. Foster, T. Goradia, and T. J. Ostrand, "Experiments of the Effectiveness of Dataflow- and Controlflow-Based Test Adequacy Criteria," in *Proc. of ICSE'94*. IEEE Computer Society / ACM Press, 1994, pp. 191–200.
[18] A. Groce, S. Chaki, D. Kroening, and O. Strichman, "Error Explanation with Distance Metrics," *STTT*, vol. 8, no. 3, pp. 229–247, 2006.
[19] G. Gößler, D. L. Métayer, and J. Raclet, "Causality Analysis in Contract Violation," in *Proc. of RV'10*, ser. LNCS, vol. 6418. Springer, 2010, pp. 270–284.
[20] G. Gößler and D. L. Métayer, "A General Trace-Based Framework of Logical Causality," in *Proc. of FACS'13*, ser. LNCS, vol. 8348. Springer, 2013, pp. 157–173.
[21] A. Beer, S. Heidinger, U. Kühne, F. Leitner-Fischer, and S. Leue, "Symbolic Causality Checking Using Bounded Model Checking," in *Proc. of SPIN'15*, ser. LNCS, vol. 9232. Springer, 2015, pp. 203–221.
[22] H. Jin, K. Ravi, and F. Somenzi, "Fate and Free Will in Error Traces," in *Proc. of TACAS'02*, ser. LNCS, vol. 2280. Springer, 2002, pp. 445–459.
[23] T. Ball, M. Naik, and S. K. Rajamani, "From Symptom to Cause: Localizing Errors in Counterexample Traces," in *Proc. of POPL'03*. ACM, 2003, pp. 97–105.
[24] A. Groce and W. Visser, "What Went Wrong: Explaining Counterexamples," in *Proc. of SPIN'03*, ser. LNCS, vol. 2648. Springer, 2003, pp. 121–135.
[25] K. Ravi and F. Somenzi, "Minimal Assignments for Bounded Model Checking," in *Proc. of TACAS'04*, ser. LNCS, vol. 2988. Springer, 2004, pp. 31–45.
[26] S. Leue and M. T. Befrouei, "Mining Sequential Patterns to Explain Concurrent Counterexamples," in *Proc. of SPIN'13*, ser. LNCS, vol. 7976. Springer, 2013, pp. 264–281.
[27] M. T. Befrouei, C. Wang, and G. Weissenbacher, "Abstraction and Mining of Traces to Explain Concurrency Bugs," in *Proc. of RV'14*, ser. LNCS, vol. 8734. Springer, 2014, pp. 162–177.
[28] X. Yan, J. Han, and R. Afshar, "CloSpan: Mining Closed Sequential Patterns in Large Databases," in *Proc. of SDM'03*. SIAM, 2003, pp. 166–177.
[29] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A Survey on Software Fault Localization," *IEEE TSE*, vol. 42, no. 8, pp. 707–740, 2016.
[30] D. Beyer, A. Chlipala, T. A. Henzinger, R. Jhala, and R. Majumdar, "Generating Tests from Counterexamples," in *Proc. of ICSE'04*. IEEE Computer Society, 2004, pp. 326–335.
[31] B. Baudry, F. Fleurey, and Y. L. Traon, "Improving Test Suites for Efficient Fault Localization," in *Proc. of ICSE'06*. ACM, 2006, pp. 82–91.
[32] M. Papadakis and Y. L. Traon, "Effective Fault Localization via Mutation Analysis: A Selective Mutation Approach," in *Proc. of SAC'14*. ACM, 2014, pp. 1293–1300.

Gianluca Barbon Gianluca Barbon received a Master degree in Computer Science from the Ca' Foscari University of Venice in 2015. He received his PhD from the Université Grenoble Alpes in 2018.

Vincent Leroy Vincent Leroy is an associate Professor at the Université Grenoble Alpes since 2012. Before that, he was a post-doctorate researcher at Yahoo! Research. He is interested in distributed system and large-scale data analysis. He received his PhD in Computer Science from Inria in 2010.

Gwen Salaün Gwen Salaün received a PhD degree in Computer Science from the University of Nantes (France) in 2003. He was associate professor at Ensimag / Grenoble INP (France) from 2009 to 2016. He is currently full professor at the Université Grenoble Alpes. He is interested in formal methods, automated verification, distributed systems and software engineering.