Debugging of Concurrent Systems using Counterexample Analysis

Gianluca Barbon[1], Vincent Leroy[2], and Gwen Salaün[1]

[1] Univ. Grenoble Alpes, CNRS, Inria, LIG, France
[2] Univ. Grenoble Alpes, CNRS, LIG, France

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 hundreds of actions, (ii) the debugging task is mostly achieved manually, and (iii) the counterexample does not give any clue on the state of the system (*e.g.*, parallelism or data expressions) when the error occurs. This paper 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. To do so, we first extract in the model all the counterexamples. Second, we define an analysis algorithm that identifies actions that make the behaviour skip from incorrect to correct behaviours, making these actions relevant from a debugging perspective. Our approach is fully automated by a tool that we implemented and applied on real-world case studies from various application areas for evaluation purposes.

1 Introduction

Concurrent and distributed applications are used in various domains, such as cyber-physical systems, software and middleware technologies, Service Oriented Computing, cloud computing, or the Internet of Things. The design and development of these applications is complex and cannot be achieved without introducing subtle bugs, which are defects of the software that prevent the correct behaviour of the system. The process of finding and resolving bugs is commonly called *debugging*. This process is 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. There is a need for automatic techniques that can help the developer in detecting and understanding those bugs.

Model checking [8] 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 paper, 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 give any clue on the state of the system (*e.g.*, parallelism or data expressions) when the error occurs.

This work aims at developing a new approach for simplifying the comprehension of counterexamples and thus favouring usability of model checking techniques. In order 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 better identify actions that caused the bug. The goal of our approach is to return as result an abstraction of counterexamples, which contains only those actions.

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. Second, we define an analysis algorithm that identifies actions at the frontier between the new LTS and the original one. The frontier represents the area where counterexamples and correct behaviours, that share a common prefix, split in different paths. Actions at the frontier are relevant since they are responsible for the choice between a correct behaviour and a counterexample. We have implemented our approach in a tool and validated it on a set of real-world case studies from various application areas. Our experiments show that our approach is able to reduce the size of counterexamples by keeping only relevant actions at the frontier, and thus making the debugging process easier.

The rest of this paper is organized as follows. Section 2 introduces LTS models and model checking notions. Section 3 presents our counterexample abstraction techniques, including the generation of the LTS containing all the counterexamples and the process for identifying relevant actions in counterexamples. In Section 4, we describe our implementation and we apply it on real-word examples. Section 5 presents related work while Section 6 concludes this paper.

2 Preliminaries

In this work, we adopt *Labelled Transition Systems (LTS)* as behavioural models 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 [7] and compilers from the CADP toolbox [11] for obtaining LTSs from LNT specifications (see Section 4 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 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 $t_\delta(M) = \emptyset$.

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 [2]. 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 [16] in Section 4). It can be semantically characterized 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 algorithm, which takes its foundation into the notion of preorder simulation between two LTSs [19].

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 Analysis

In this section, we describe our approach to simplify counterexamples. 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 match all states of the counterexample LTS with states of the full LTS. This step 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. We define a notion of *neighbourhood* to extract sets of relevant transitions at the frontier and a procedure to collect the set of all neighbourhoods. Finally, by keeping transitions in these neighbourhoods, we are able to provide an abstraction of a given counterexample. To sum up, our approach consists of the four following steps, that we detail in the rest of this section:

1. Counterexample LTS generation
2. States matching
3. States comparison
4. Counterexample abstraction

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 belongs to $t(M_C)$ are not counterexamples. Moreover, traces in the counterexample LTS share prefixes with correct traces in the full 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:

1. 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

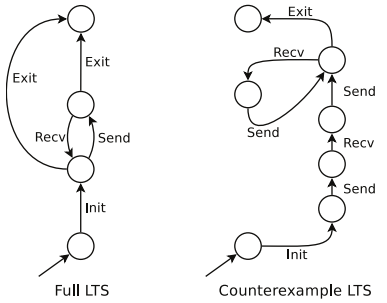

Fig. 1. Full LTS and counterexample LTS

described in [12]. 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 [1] 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 [17]. 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.

2. 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 $t_\delta(M_\varphi) = t_\varphi \cap \Sigma_F^*$.
3. Pruning of the useless transitions generated during the previous step. In particular, we use the pruning algorithm proposed in [15] to remove the traces produced by the synchronous product that are not the prefix of any final trace.

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

Let us illustrate this algorithm 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 the examples.

3.2 States Matching

We now need to match each state belonging to the counterexample LTS with the states from the full LTS. To do this, we define a matching relation between each state of the two LTSs, by relying on the simulation relation introduced in

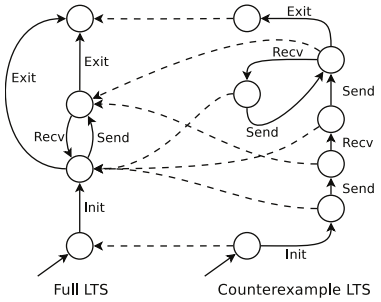

Fig. 2. States matching

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. The algorithm that we have implemented to build the simulation between M_C and M_F relies on well-known graph traversal algorithms. More precisely, it relies on Breadth-First Search (BFS) to explore the graph. The algorithm is capable of performing backtracking steps in case it cannot match some states (this may happen due to nondeterministic behaviours present in both LTSs).

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 2. Note that multiple states of the counterexample LTS may correspond to a single state of the full LTS. In the example of Figure 2, the property φ has become unsatisfied after several iterations of the loop composed of *Send* and *Recv* actions, so that loop has been partially rolled out in the counterexample LTS, resulting in a correspondence of several states of the counterexample LTS to a single state of the full LTS.

It may also occur that a single state of the counterexample LTS may correspond to multiple states of the full LTS. For instance, the example given in Figure 3 shows a full LTS and a counterexample LTS produced with a property that avoids *Recv* actions after a *Send* action. Thus, there exists a correspondence of more than one state of the full LTS with a single state of the counterexample LTS. In this specific case, the counterexample LTS can be described using a single trace, since the two states with an exiting *Send* transition after the *Init* transition simulate only one state in the counterexample LTS.

3.3 States Comparison

The result of the matching algorithm is then analysed in order 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*.

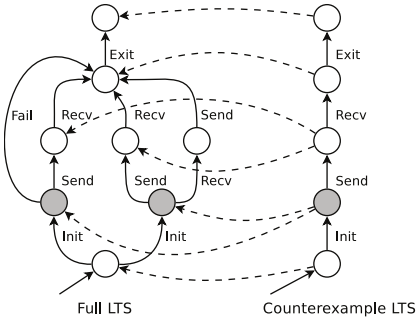

Fig. 3. Multiple matching

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. We call these transitions *relevant predecessors*. Correct transitions allow us to introduce the notion of *frontier*. The frontier is a set of states at the border between the counterexample LTS and the rest of the full LTS, where for two matched states, there exists a correct transition in the full LTS.

Definition 8. *(Frontier) 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*, the frontier is the set of states* $S_{fr} \subseteq S_F$ *such that for each* $s \in S_{fr}$*, there exists* $s' \in S_C$*, such that* $s' \sqsubseteq s$ *and there exists a correct transition* $s \xrightarrow{l} s'' \in T_F$*.*

A given state in the frontier allows us in a second step to identify a *neighbourhood* in the corresponding counterexample LTS, which consists of all incoming and outgoing transitions of that state.

Definition 9. *(Neighbourhood) Given an LTS* $M_F = (S_F, s_F^0, \Sigma_F, T_F)$*, a property* φ*, the counterexample LTS* $M_C = (S_C, s_C^0, \Sigma_C, T_C)$*, two states* $s \in S_{fr}$ *and* $s' \in S_C$ *such that* $s' \sqsubseteq s$*, the neighbourhood of state* s' *is the set of transitions* $T_{nb} \subseteq T_C$ *such that for each* $t \in T_{nb}$*, either* $t = s'' \xrightarrow{l} s' \in T_C$ *or* $t = s' \xrightarrow{l} s'' \in T_C$*.*

Let us illustrate these notions 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 is at the frontier, thus it has been matched by a state of the counterexample LTS on the right hand side and it has correct transitions outgoing from it. The incoming and outgoing transitions for this state in the counterexample LTS correspond to the neighbourhood.

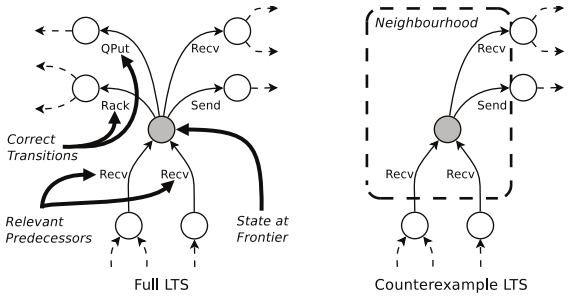

Fig. 4. Example of neighbourhood

3.4 Counterexample Abstraction

The final goal is to abstract a counterexample of the model in order to highlight the source of the bug and thus favour the comprehension of its cause. Given the counterexample LTS M_C, produced from a model M_F and a property φ, where neighbourhoods have been identified in the previous subsection, 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_C.
2. Identification of states in c_e that are matched to states in M_C, which belong to a neighbourhood.
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 on the top side of Figure 5. Once the set of neighbourhoods in the counterexample 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 simplified counterexample shown on the bottom side of the figure. We will comment on the relevance and benefit of these results on real-world examples in the next section.

4 Tool Support

In this section, we successively present the implementation of our approach, illustrate it on a case study, and present experimental results on examples found in the literature.

4.1 Implementation

Our tool is depicted in Fig. 6 and consists of two main parts. The first one implements the counterexample LTS generation step described in Section 3.1. It relies

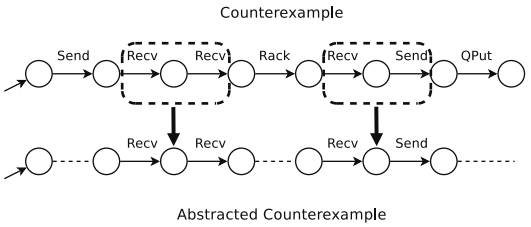

Fig. 5. Counterexample abstraction

on the CADP toolbox [11], which enables one to specify and analyse concurrent systems using model and equivalence checking techniques. We particularly make use of the LNT value passing process algebra [7] for specifying systems, of the BCG binary format for representing LTSs, and of the MCL mu-calculus logic [16] for describing safety temporal properties. The LNT specification is automatically transformed into an LTS model in BCG format (the full LTS in Section 3) using CADP compilers. The CADP model checker (Evaluator [16]) takes as input an MCL property and an input specification/model (LNT or LTS), and returns a verdict (true or false + a counterexample if the property is violated). The computation of the counterexample LTS is achieved by a script we wrote using SVL [10], 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 [12]; EXP.OPEN for building LTS products; Reductor for minimizing LTSs; Scrutator [15] for removing spurious traces in LTSs.

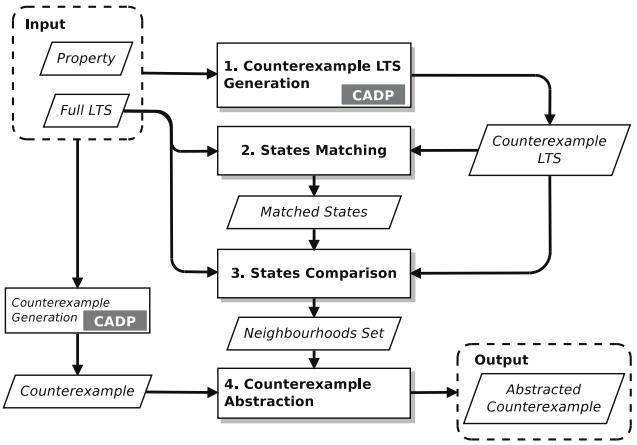

Fig. 6. Overview of the tool support

The second part of our tool implements the algorithms for state matching (2), state comparison (3) and counterexample abstraction (4), described from Section 3.2 to Section 3.4. This part of the tool has been implemented in Java and consists of about 2,500 lines of code. The tool takes as input the files containing the full and the counterexample LTS, converted into an intermediate ASCII format called AUT (provided by CADP), and stores them in memory using a Java graph modelling library. The matching step (2) is based on a BFS graph search algorithm in order to build the simulation relation between the two LTSs. The state matching is then stored into a map, used by the state comparison step (3) to analyse outgoing transitions for each association of states between the two LTSs. This allows us to retrieve the set of neighbourhoods. Finally, the counterexample abstraction step (4) first produces the shortest counterexample from the full LTS and the property by using the Evaluator model checker, and second performs the counterexample reduction by locating and keeping actions that correspond to neighbourhoods. The result retrieved by our tool consists of the shortest counterexample abstracted in the form of a list of sub-sequences of actions, accompanied by the list of all neighbourhoods.

4.2 Case Study

We now describe an example taken from a real-world case study [20]. The example models a sanitary agency that aims at supporting elderly citizens in receiving sanitary agency 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 7 gives the LTS model for each participant. We assume in this example that the participants interact together asynchronously by exchanging messages via FIFO buffers.

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:

$$[\ \mathsf{true}^*\ .\ \text{'REQUEST_EM'}\ .\ \mathsf{true}^*\ .\ \text{'PAYMENTT_EM'}\ .\ \mathsf{true}^*\ ]\ \mathsf{false}$$

We applied our tool to the sanitary agency model with the aforementioned property. Our tool was able to identify seven neighbourhoods in the couterexample LTS. The shortest counterexample involves three neighbourhoods, and this allows us to reduce its size from 19 actions to only 6 actions. Figure 8 shows (from left to right) the full LTS of the sanitary agency model, the shortest counterexample, and the three neighbourhoods (+ correct transitions) for this counterexample. 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

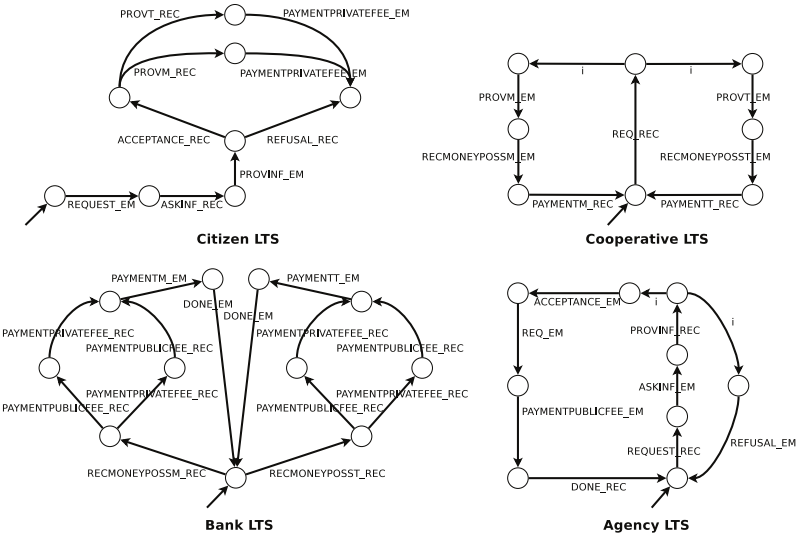

Fig. 7. LTS models for the sanitary agency

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. Indeed, when a citizen request is refused by the sanitary agency, the execution skips the part of the system behaviour where the transportation service and payment appear. The two next neighbourhoods pinpoint the second reason of property violation. They show that actions RECMONEYPOST_EM and PROVT_EM have been performed, which correspond to triggering the request for payment of the transportation service, that is not permitted by the property.

Our solution thus allows the developer to identify the cause of the property violation by identifying specific actions in counterexamples via the notion of neighbourhood. It is worth stressing that, since our approach applies on the counterexample LTS and computes all the neighbourhoods, the returned solution is able to pinpoint all the causes of the property violation, as we have shown with the example above.

4.3 Experimental Results

We carried out experiments on about 20 real-world examples found in the literature. For each example, we use as input an LNT specification or an LTS model as well as a safety property. Table 1 summarizes the results for some of these experiments. The first two columns contain the name of the model, the reference to the corresponding article, and the property. The third and fourth columns show the size of the full and the counterexample LTSs, respectively, in terms of

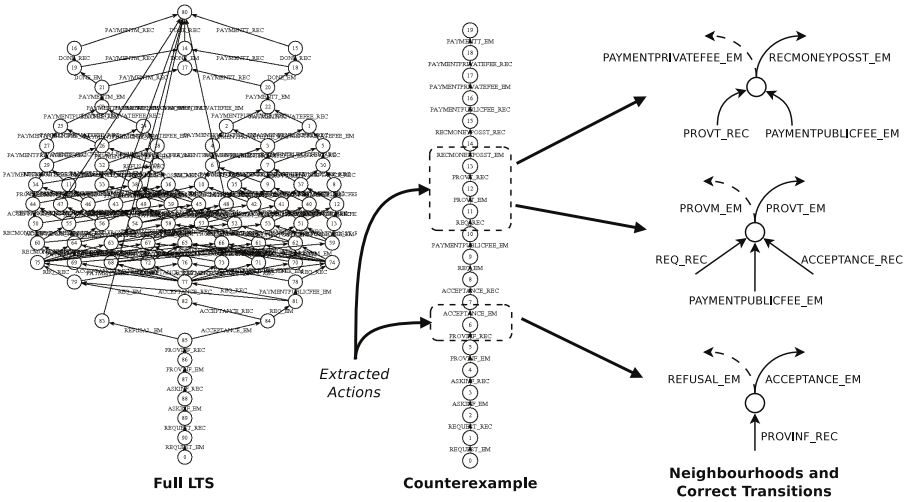

Fig. 8. Sanitary agency: full LTS and shortest counterexample

number of states, transitions and labels. The following columns give the number of identified neighbourhoods, the size of the shortest (retrieved with breadth first search techniques) and of the abstracted counterexample, respectively. Finally, the last two columns detail the execution time for the counterexample LTS production, and for the matching and comparison algorithms (in seconds).

Example	φ	L_F (s/t/l)	L_C (s/t/l)	$\|N\|$	$\|Ce\|$	$\|Ce_r\|$	t_{L_F}	t_N
sanitary agency [20]	φ_{sa1}	227 / 492 / 31	226 / 485 / 31	6	17	2	6.3s	0.3s
sanitary agency [20]	φ_{sa2}	142 / 291 / 31	492 / 943 / 31	18	64	6	5.7s	0.2s
ssh protocol [14]	φ_{sp1}	23 / 25 / 23	20 / 20 / 19	2	14	3	4.9s	0.2s
ssh protocol [14]	φ_{sp2}	23 / 25 / 23	35 / 35 / 19	4	29	7	4.8s	0.1s
client supplier [6]	φ_{cs1}	35 / 45 / 26	29 / 33 / 24	3	18	5	4.6s	0.1s
client supplier [6]	φ_{cs2}	35 / 45 / 26	25 / 25 / 24	4	19	6	4.9s	0.1s
client supplier [6]	φ_{cs3}	35 / 46 / 26	33 / 41 / 24	2	15	2	4.8s	0.2s
train station [21]	φ_{ts}	39 / 66 / 18	26 / 34 / 18	1	6	2	5.2s	0.2s
selfconfig [22]	φ_{ac}	314 / 810 / 27	159 / 355 / 27	30	14	5	5.6s	0.3s
online stock broker [9]	φ_{osb}	1331 / 2770 / 13	2653 / 5562 / 13	61	23	23	4.9s	0.7s

Table 1. Experimental results

First of all, we can see a clear gain in length between the original counterexample and the abstracted one, which keeps only relevant actions using our approach. There is one case (online stock broker, last row) in which our solution was not able to reduce the counterexample. This may occur in specific cases when the counterexample (the shortest here) does not exhibit any actions corresponding to transitions in a neighbourhood. In that particular situation,

our abstraction techniques cannot help the developer in the identification of the cause of the property violation.

As far as computation time is concerned, the table shows that, for these examples, the time for producing counterexample LTSs is slightly longer than the time for computing the matching/comparison algorithms, which is very low (less than a second). The script for counterexample LTS computation is longer because it calls several CADP tools in sequence, which takes time.

5 Related Work

In this section, we survey related papers providing techniques for supporting the debugging of specifications and programs. LocFaults [5] is a flow-driven and constraint-based approach for error localization. It takes as input a faulty 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 notion to locate errors in the faulty program. This work focuses on programs with numerical statements and relies on a constraint programming framework allowing the combination of Boolean and numerical constraints. In addition, the authors do not explicitly describe the capacity of their solution for analysing concurrent programs.

Concurrency is explicitly taken into account in [3, 4]. In [3], 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. In [4], 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 [24]. 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 [13], where the authors applied 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 [13] 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. Reasoning on traces as achieved in [3, 4, 13] 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.

Another solution for localization of faults in failing programs consists in using testing techniques. As an example, [18] presents a mutation-based fault localization approach and suggests the use of a sufficient mutant set to locate effectively the faulty statements. This mutation analysis approach applies on

C programs under validation using testing techniques whereas we focus on formal specifications and models being analysed using model checking techniques. In [23], 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.

6 Conclusion

In this paper, we have proposed a new method for debugging concurrent systems 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 erroneous transitions in the LTS, as well as an algorithm for computing them by comparing the full LTS and the LTS consisting of all counterexamples. Finally, we have implemented our approach as a tool and evaluated it on real-world case studies, showing the advantage of the counterexample abstraction in practice when adopting the neighbourhood approach.

As far as future improvements are concerned, a first perspective of this work is to extend our approach to focus on probabilistic specifications and models, and refine our LTS analysis techniques for handling those models. Another perspective is to increase the scope of system requirements that we can take into account. Indeed, although safety properties already allow us to define most requirements for real-world systems, we would like to consider liveness properties as well. Finally, we plan to investigate the introduction of code colouring in the specification by highlighting code portions that correspond to the source of the problem according to our approach.

Acknowledgements. We would like to thank Frédéric Lang and Radu Mateescu for their valuable suggestions to improve the paper.

References

1. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.
2. C. Baier and J. Katoen. *Principles of Model Checking*. MIT Press, 2008.
3. A. Beer, S. Heidinger, U. Kühne, F. Leitner-Fischer, and S. Leue. Symbolic Causality Checking Using Bounded Model Checking. In *Proc. of SPIN'15*, volume 9232 of *LNCS*. Springer, 2015.
4. M. T. Befrouei, C. Wang, and G. Weissenbacher. Abstraction and Mining of Traces to Explain Concurrency Bugs. In *Proc. of RV'14*, volume 8734 of *LNCS*. Springer, 2014.

5. M. Bekkouche, H. Collavizza, and M. Rueher. LocFaults: A New Flow-driven and Constraint-based Error Localization Approach. In *Proc. of SAC'15*. ACM, 2015.
6. J. Cámara, J. A. 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.
7. D. Champelovier, X. Clerc, H. Garavel, Y. Guerte, F. Lang, C. McKinty, V. Powazny, W. Serwe, and G. Smeding. Reference Manual of the LOTOS NT to LOTOS Translator (Version 6.1). INRIA/VASY, 131 pages, 2014.
8. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model checking*. MIT Press, 2001.
9. 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.
10. H. Garavel and F. Lang. SVL: A Scripting Language for Compositional Verification. In *Proc. of FORTE'01*, volume 197 of *IFIP Conference Proceedings*. Kluwer, 2001.
11. H. Garavel, F. Lang, R. Mateescu, and W. Serwe. CADP 2011: A Toolbox for the Construction and Analysis of Distributed Processes. *STTT*, 15(2):89–107, 2013.
12. F. Lang and R. Mateescu. Partial Model Checking using Networks of Labelled Transition Systems and Boole an Equation Systems. *Logical Methods in Computer Science*, 9(4), 2013.
13. S. Leue and M. T. Befrouei. Mining Sequential Patterns to Explain Concurrent Counterexamples. In *Proc. of SPIN'13*, volume 7976 of *LNCS*. Springer, 2013.
14. J. A. Martín and E. Pimentel. Contracts for Security Adaptation. *J. Log. Algebr. Program.*, 80(3-5), 2011.
15. R. Mateescu, P. Poizat, and G. Salaün. Adaptation of Service Protocols Using Process Algebra and On-the-Fly Reduction Techniques. *IEEE Trans. Software Eng.*, 38(4):755–777, 2012.
16. R. Mateescu and D. Thivolle. A Model Checking Language for Concurrent Value-Passing Systems. In *Proc. of FM'08*, volume 5014 of *LNCS*. Springer, 2008.
17. R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.
18. M. Papadakis and Y. L. Traon. Effective Fault Localization via Mutation Analysis: A Selective Mutation Approach. In *Proc. of SAC'14*. ACM, 2014.
19. D. M. R. Park. Concurrency and Automata on Infinite Sequences. In *Proc. of the 5th Theoretical Computer Science Conference*, volume 104 of *LNCS*. Springer, 1981.
20. 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.
21. G. Salaün, T. Bultan, and N. Roohi. Realizability of Choreographies Using Process Algebra Encodings. *IEEE Transactions on Services Computing*, 5(3):290–304, 2012.
22. G. Salaün, X. Etchevers, N. D. Palma, F. Boyer, and T. Coupaye. Verification of a Self-configuration Protocol for Distributed Applications in the Cloud. In *Assurances for Self-Adaptive Systems*, pages 60–79. Springer, 2013.
23. G. Salaün and L. Ye. Debugging Process Algebra Specifications. In *Proc. of VMCAI'15*, volume 8931 of *LNCS*. Springer, 2015.
24. X. Yan, J. Han, and R. Afshar. CloSpan: Mining Closed Sequential Patterns in Large Databases. In *Proc. of SDM'03*. SIAM, 2003.