Counterexample Simplification for Liveness Property Violation

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

[1] Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG, 38000 Grenoble, France
[2] Univ. Grenoble Alpes, CNRS, Grenoble INP, LIG, 38000 Grenoble, France

Abstract. Model checking techniques verify 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 because the developer has to understand by manual analysis all the steps (possibly many) that have provoked the bug. The objective of this work is to improve the comprehension of counterexamples and thus to simplify the detection of the source of the bug. Given a liveness property, our approach first extends the model with prefix / suffix information w.r.t. that property. This enriched model is then analysed to identify specific states called neighbourhoods. A neighbourhood consists of a choice between transitions leading to a correct or incorrect part of the model. We exploit this notion of neighbourhood to highlight relevant parts of the counterexample, which makes easier its comprehension. Our approach is fully automated by a tool that we implemented and that was validated on several real-world case studies.

1 Introduction

Recent computing trends promote the development of hardware and software applications that are intrinsically parallel, distributed, and concurrent. This is the case of service-oriented computing, cloud computing, cyber-physical systems or the Internet of Things. Designing and developing distributed software in this context is a tedious and error-prone task, and the ever increasing software complexity is making matters even worse. Although we are still far from proposing techniques and tools avoiding the existence of bugs in a software under development, we know how to automatically chase and find bugs that would be very difficult, if not impossible, to detect manually.

Model checking [1] is an established technique for automatically verifying that a behavioural model, e.g., a Labelled Transition System (LTS), satisfies a given temporal formula written with temporal logic. 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 many actions, (ii) the

debugging task is mostly achieved manually, and (iii) the actions in the counterexample seem to have the same importance even if it is not the case. Note that in this work, we have a specific focus on liveness properties, and more precisely inevitability properties which are one of the classes of liveness properties most used by developers in practice [9].

Our goal in this paper is to simplify the debugging of concurrent systems whose specification compiles into a behavioural model. To do so, we propose a novel approach for improving the comprehension of counterexamples by highlighting some of the actions in the counterexample that are of prime importance, that is, actions that make the specification go from a (potentially) correct behaviour to an incorrect one. These parts of the model correspond to decisions or choices that are of particular interest because they might explain the source of the bug. Once these specific actions have been identified, they can be used for building a simplified version of the counterexample, keeping only actions that are relevant from a debugging perspective.

Our approach takes as input a behavioural model (LTS) describing all possible executions of a system. This LTS can be obtained by compilation from a higher-level textual specification language such as process algebra. Given such an LTS and a liveness property, in a first step, we enhance each state of the LTS model with prefix/suffix information about the actions belonging to the property that have already been or remain to be executed. This enriched LTS is then analysed to identify specific states called neighbourhoods. A neighbourhood consists of a choice between transitions leading to a correct or incorrect part of the model. Those states identify specific parts of the specification that may explain the appearance of the bug and are therefore meaningful from a debugging perspective. Several simplification techniques can be defined on top of this notion of neighbourhood, which aim at removing irrelevant parts of the counterexample and highlighting relevant ones to simplify its comprehension. Our approach is fully automated by a tool that we implemented. This tool was applied on several real-world case studies for evaluation purposes.

The paper is organized as follows. Section 2 introduces behavioural model and model checking. Section 3 presents the technique for computing the LTS enriched with prefix/suffix information. This information is then used for identifying neighbourhoods and building counterexample abstractions from them. Section 4 illustrates our approach on two real-world case studies. Section 5 overviews related work and Section 6 concludes this paper.

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 state identifiers;* $s^0 \in S$ *is the initial state identifier;* 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 [6] and LOTOS [5] as specification languages and compilers from the CADP toolbox [10] for obtaining LTSs from these specifications. 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 $s^0 \xrightarrow{l_1} s_1 \in T, s_1 \xrightarrow{l_2} s_2 \in T, \ldots, s_{n-1} \xrightarrow{l_n} s_n \in T$. A trace is either infinite because of loops or the last state s_n has no outgoing transitions. The set of all traces of M is written as $t(M)$.*

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 [1]. In this work, we focus on a class of liveness properties, called *inevitable execution* properties. Most of the patterns that commonly occur in the specification of liveness properties make use of the inevitable executions. This is the case of the *Response Property Pattern*, that is the most common pattern in [9]. An inevitable execution property states that, given an LTS M and an action l, every trace from the initial state in M presents a transition with the action l. In this work we support *nested inevitable executions*. For instance, a property with two nested actions l_1 and l_2 states that every trace in a given model must exhibit the action l_1 later followed by the action l_2. Note that the two actions do not need to be contiguous in traces. To express nested inevitable executions we define a *nested inevitability operator* using the *Action-based Computation Tree Logic (ACTL)* [8]:

Definition 3. *(Nested Inevitability Operator) Given a sequence of labels $l_1, \ldots l_n$, the nested inevitability operator is defined as*

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

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

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

Definition 4. *(Counterexample) Given an LTS $M = (S, s^0, \Sigma, T)$ and a property φ, a counterexample is any trace which belongs to $t(M) \setminus t_\varphi$. A counterexample can be in the form of an elementary trace, which is a trace where states*

are pairwise distinct, or a lasso, which is a trace $s^0 \xrightarrow{l_1} s_1 \in T, \ldots, s_{n-2} \xrightarrow{l_{n-1}} s_{n-1} \in T, s_{n-1} \xrightarrow{l_n} s_n \in T$*, such that* $s^0 \xrightarrow{l_1} s_1 \in T, \ldots, s_{n-2} \xrightarrow{l_{n-1}} s_{n-1} \in T$ *is an elementary trace and* $s_n = s_i$ *for some* $0 \leq i < n$*.*

3 Counterexample Simplification

In this section we discuss in detail our approach to simplify counterexamples. Section 3.1 presents the notions of prefixes and suffixes. Section 3.2 describes the algorithm to compute them and enrich the initial LTS. In Section 3.3 we identify transitions and we introduce the neighbourhood notion. Section 3.4 presents simplification techniques, focusing on one of them.

3.1 Prefixes and Suffixes

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

Definition 5. *(Sequence of Inevitable Actions) Given an inevitable execution property* $p = \texttt{Inev}(l_1, \ \ldots, \ l_n)$*, the sequence of concatenated labels* $k = l_1 \cdot l_2 \cdot \ldots \cdot l_n$ *of size* $n \in \mathbb{N}$ *is the sequence of inevitable actions that respect the order defined by the nested inevitability operator.*

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

Definition 6. *(Matching Operator) Given an LTS* $M = (S, s^0, \Sigma, T)$*, a sequence of labels* $j = a_1 \cdot a_2 \cdot \ldots \cdot a_n$*, a sequence of contiguous transitions* $z = s_1 \xrightarrow{l_1} s_2 \in T, s_2 \xrightarrow{l_2} s_3 \in \ T, \ldots, s_{m-1} \xrightarrow{l_{m-1}} s_m \in T$*, z is said to match j, written* $j \prec z$*, if there exists integers* $1 \leq i_1 < i_2 < \ldots < i_n \leq m$ *such that* $a_1 = l_{i_1}, \ a_2 = l_{i_2}, \ \ldots, \ a_n = l_{i_n}$*.*

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

Definition 7. *(Max and Common Prefix) Given an LTS $M = (S, s^0, \Sigma, T)$, a sequence of inevitable actions k, the set $\mathbb{P}^k$ of all the prefixes of k, a state $s \in S$, the max prefix, defined as mp_s, is the longest element in $\mathbb{P}^k$ such that $\exists t \in \mathbb{T}^e_s,\ mp_s \prec t$. The common prefix, defined as cp_s, is the longest element in $\mathbb{P}^k$ such that $\forall t \in \mathbb{T}^e_s,\ cp_s \prec t$.*

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

Definition 8. *(Max and Common Suffix) Given an LTS $M = (S, s^0, \Sigma, T)$, a sequence of inevitable actions k, the set $\mathbb{S}^k$ of all the suffixes of k, a state $s \in S$, the max suffix, defined as ms_s, is the longest element in $\mathbb{S}^k$ such that $\exists t \in \mathbb{T}^o_s,\ ms_s \prec t$. The common suffix, defined as cs_s, is the longest element in $\mathbb{S}^k$ such that $\forall t \in \mathbb{T}^o_s,\ cs_s \prec t$.*

The example given in Figure 1 shows the max/common prefixes and suffixes calculated on each state of an LTS for a given sequence of inevitable actions $k = A \cdot Y$. Let us take a look at state 8: the *cp* value shows that the action A exists in every prefix of k produced by prefixes of traces that end in state 8. Conversely, the *cs* value in state 8 is empty while the *ms* value is $A \cdot Y$, meaning that the suffix $A \cdot Y$ is not contained in every suffix of traces that starts in state 8. As a matter of fact, we can see that the only suffix of traces that respects the k sequence is the one that begins with the transition $9 \xrightarrow{C} 10 \in T$.

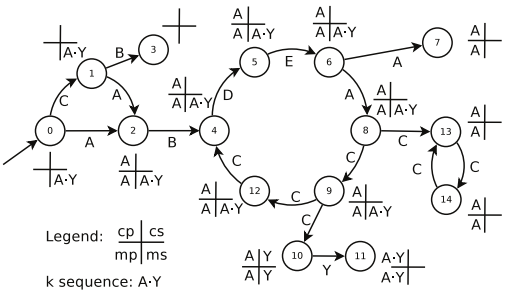

Fig. 1. Max / common prefixes and suffixes

In some cases inevitable execution properties might not be satisfied because of loops in which the execution of the system can remain infinitely. Our notion of suffix allows us to discover such loops and understand whether they prevent the satisfaction of the property. One of these loops is present in the example

in Figure 1 and is composed of states 4, 5, 6, 8, 9 and 12. These loops are treated in the next section by extracting the *Strongly Connected Components (SCCs)* [17] from the LTS. The LTS with the max/common prefixes (suffixes, resp.) computed for each state is called *enriched LTS*.

Definition 9. *(Enriched LTS) Given an LTS* $M = (S, s^0, \Sigma, T)$ *and a sequence of inevitable actions* k*, the enriched LTS is a tuple* $M_E^k = (S_E, s_E^0, \Sigma_E, T_E)$ *such that each state* $s_E \in S_E$ *is a tuple* $s_E = (s, mp_s, cp_s, ms_s, cs_s)$*, where* $s \in S$*,* $mp_s, cp_s \in \mathbb{P}_s^k$*,* $ms_s, cs_s \in \mathbb{S}_s^k$*;* $s_E^0 = (s^0, mp_{s^0}, cp_{s^0}, ms_{s^0}, cs_{s^0})$*;* $\Sigma_E = \Sigma$*;* $T_E \subseteq S_E \times \Sigma_E \times S_E$*, where* $\forall s \xrightarrow{l} s' \in T$*,* $(s, mp_s, cp_s, ms_s, cs_s) \xrightarrow{l} (s', mp_{s'}, cp_{s'}, ms_{s'}, cs_{s'}) \in T_E$*.*

3.2 Prefixes and Suffixes Calculation

This section presents the computation of the prefixes and suffixes defined in Section 3.1. In order to handle cycles in an LTS we use the notion of SCC, that is a partition of an LTS where every state is reachable from any other state. Note that every SCC in an LTS is also a Maximally Strongly Connected Component, since an SCC cannot be subsumed by a larger SCC by definition. To detect all the SCCs in an LTS we use the Tarjan's SCCs algorithm [17]. Given an LTS $M = (S, s^0, \Sigma, T)$, the algorithm allows the detection of all the SCCs in linear time, with a cost of $O(|S| + |T|)$. Given a sequence of inevitable actions k, our approach considers each SCC of the LTS, and computes the max/common prefixes and suffixes for every state in the SCC. Note that we start computing prefixes for states in a given SCC only when all its predecessors SCCs have been computed (in the case of suffixes we first compute all the successors).

We now introduce some notions related to the SCCs that we will use throughout the whole section. Given an LTS $M = (S, s^0, \Sigma, T)$ and an SCC G in M, where the sets of states and transitions in G are defined as S_G and T_G, respectively, we denote as $S_G^e \subseteq S_G$ the set of *initial states* of G, such that, given a transition $s \xrightarrow{l} s' \in T$, the state $s \notin S_G$ and $s' \in S_G^e$. The transition $s \xrightarrow{l} s' \in T$ is defined as *incoming transition* and the set of incoming transitions is written as T_G^e. Similarly, we denote as $S_G^o \subseteq S_G$ the set of *outgoing states* such that, given a transition $s \xrightarrow{l} s' \in T$, the state $s \in S_G^o$ and $s' \notin S_G$. The transition $s \xrightarrow{l} s' \in T$ is defined as an *outgoing transition* and the set of outgoing transitions is written as T_G^o. We denote as $\mathbb{G}_\mathbb{M}$ the *component graph* [7] of an LTS M where the states are given by SCCs of M. The SCC containing the initial state s_0 of the LTS M does not have any predecessors and it is defined as G^0. By definition, since all cycles are contained in SCCs, $\mathbb{G}_\mathbb{M}$ is a directed acyclic graph. The rest of this section presents the computation of the max prefix (suffix, resp.) and of the common prefix (suffix, resp.) for states of an SCC.

Max Prefix Calculation. The max prefix inside an SCC is computed by first extracting the longest max prefix among the incoming states of the SCC. Second, the incoming max prefix is extended with actions contained inside the SCC to produce the longest (possible) prefix of k. Note that the max prefix is the same

for all the states of an SCC. The cost of the computation for an SCC G is $O(|T_G^e| + |T_G| + |k|)$, since we first have to explore all the incoming transitions to compute the initial max prefix, and second we have to collect all the actions in the SCC that are also present in k. Let us consider the SCC composed of states 1, 2 and 3 in Figure 2 (note that SCCs in states 0, 4 and 5 are trivial). Given $k = A \cdot B \cdot C$, the initial max prefix for the SCC is A, since the transition from state 0 to state 1 is the only incoming transition and it contains the first action of the k sequence. One can notice that by looping inside the SCC it is possible to complete the k sequence, since the SCC contains also actions B and C. Consequently, the max prefix in each state of the SCC (states 1, 2 and 3) is equivalent to the k sequence.

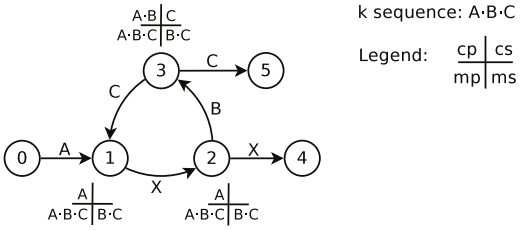

Fig. 2. Prefix and suffix calculation on an SCC

Max Suffix Calculation. The max suffix is computed similarly to the max prefix, by considering suffixes of successors instead of prefixes of predecessors. In the example in Figure 2 the max suffix for every state in the SCC is $B \cdot C$, since they are the only two actions contained in the SCC that also exist in k.

Common Prefix Calculation. We describe here the computation of the common prefix for each state in an SCC. The pseudo-code of this procedure is detailed in Algorithm 1. The algorithm is divided into two main steps: initialisation and internal transitions computation.

Initialisation step. Given an SCC G the algorithm initialises the common prefix of states in G to k (Line 3). There are two exceptions to this rule. First, the initial state of the LTS s^0 is initialised to the empty sequence since it has no predecessors. Second, if s is an initial state of G, cp_s is initialised with the common prefixes of its incoming transitions (Line 7). Let us take a look at the example in Figure 2. The initialisation step assigns $cp = A \cdot B \cdot C$ to states 2 and 3, while it assigns $cp = A$ to state 1, which is the only initial state of the SCC.

Internal transitions computation step. After the initialisation step, there may still be paths within G that can produce a prefix smaller than the ones in initial states. This step deals with internal transitions to detect smaller prefixes. First of all, we use Q (Line 8) as sorted set to order by increasing common prefix size the states in S_G. When modifying the common prefix of a state s, UpdatePosition(Q, s) updates the position of s in Q. The loop of Line 9 iterates on Q, removing

the first element s at each iteration. The common prefix of all successors s' of s within G is updated using a function LCP, which computes the longest common prefix between two sequences of actions. When producing a smaller prefix, the position of s' in Q is updated. Let us consider again the SCC in Figure 2. The internal transitions computation step corrects the values of cp in states 2 and 3, assigning respectively $cp = A$ and $cp = A \cdot B$. The value of cp in state 1 remains the same, since it was already the lower one among all the states of the SCC.

Correctness and complexity. The common prefix algorithm behaves similarly to the Dijkstra's algorithm that deals with the single-source shortest-paths problem in weighted directed graphs (in particular, to the implementation which uses a Fibonacci heap as priority queue). The complexity of the algorithm is $O(|T_G^e| + |T_G| + |S_G| \log |S_G|)$ because the while loop performs exactly $|S_G|$ iterations and the cost of inserting an element to Q and updating its position is $O(\log |Q|)$.

Algorithm 1 Common Prefix Computation

```
1: procedure CommonPrefix(G, k)
2:     for all s ∈ S_G do
3:         cp_s ← k
4:         if s = s^0 then cp_s ← ∅
5:         else if s ∈ S_G^e then
6:             for all s' -l-> s ∈ ExtractIncomingTrans(s) do
7:                 if s' ∉ S_G then cp_s ← LCP(cp_s, cp_s' · l)
8:     Q ← S_G
9:     while Q ≠ ∅ do
10:        s ← PopFirst(Q)
11:        for all s -l-> s' ∈ T_G do
12:            t ← LCP(cp_s · l, cp_s')
13:            if |t| < |cp_s'| then cp_s' ← t ; UpdatePosition(Q, s')
```

Common Suffix Calculation. The common suffix calculation is similar to the prefix case, but it differs in the initialisation step. In the suffix case the execution may loop into the current SCC and never go through an outgoing transition, and a state may thus have a smaller common suffix than all states from its successor SCCs. This initialisation step is presented in Algorithm 2. In the case of a final state, the suffix is empty (Line 3). Otherwise, the common suffix is initialised to the smallest suffix of k traversed by a loop from s to itself (Line 4). In the absence of loops (SCC with single state and no self-loop), MinSuffixLoop returns k. The remainder of the computation is similar to Algorithm 1, using a function LCS, which computes the longest common suffix, instead of LCP. Searching the smallest-suffix loop for each state is done by iteratively removing labels from k and looking for isolated vertices. Hence, the overall cost of the computation is $O(|T_G^o| + |k| \times (|T_G| + |S_G|) + |S_G| \log |S_G|)$. Let us consider the example in Figure 2. The initialisation step assigns $cs = B \cdot C$ to state 1, which is the smallest suffix of k that can be produced inside the SCC starting from state 1.

It then assigns the empty sequence and $cs = C$ to states 2 and 3, since they are outgoing states. The algorithm will later update the value of cs in state 1 to the empty sequence with the internal transitions computation step.

Algorithm 2 Common Suffix Computation (Initialisation Step)

```
1: procedure CommonSuffixInit(G, k)
2:     for all s ∈ S_G do
3:         if ∄s --l--> s' then cs_s ← ∅
4:         else cs_s ← MinSuffixLoop(s, k, G)
5:             if s ∈ S_G^o then
6:                 for all s --l--> s' ∈ ExtractOutgoingTrans(s) do
7:                     if s' ∉ S_G then cs_s ← LCS(cs_s, l · cs_s')
```

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

3.3 Neighbourhoods

The enriched LTS with max/common prefixes and suffixes can now be used to characterise its transitions. A transition is typed as *correct* if it always leads to a correct part of the model, as *incorrect* if it always leads to an incorrect part of the model, as *neutral* if none of the previous cases apply.

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

Definition 10. *(Correct Transition) Given an enriched LTS* $M_E^k = (S_E, s_E^0, \Sigma_E, T_E)$*, two states* $s_E = (s, mp_s, cp_s, ms_s, cs_s) \in S_E$*,* $s'_E = (s', mp_{s'}, cp_{s'}, ms_{s'}, cs_{s'}) \in S_E$*, a correct transition is a transition* $s_E \xrightarrow{l} s'_E \in T_E$ *such that* $cp = cp_s \cdot l$ *if* $cp_s \cdot l$ *is a prefix of* k*,* $cp = cp_s$ *otherwise, and* $|cp| + |cs_{s'}| \geq |k|$*.*

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

Definition 11. *(Incorrect Transition) Given an enriched LTS* $M_E^k = (S_E, s_E^0, \Sigma_E, T_E)$ *two states* $s_E = (s, mp_s, cp_s, ms_s, cs_s) \in S_E$, $s'_E = (s', mp_{s'}, cp_{s'}, ms_{s'}, cs_{s'}) \in S_E$, *an incorrect transition is a transition* $s_E \xrightarrow{l} s'_E \in T_E$ *such that* $mp = mp_s \cdot l$ *if* $mp_s \cdot l$ *is a prefix of* k, $mp = mp_s$ *otherwise, and* $|mp| + |ms_{s'}| < |k|$.

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

The LTS where correct, incorrect and neutral transitions have been detected allows us to recognise *neighbourhoods* in states in which an incoming neutral transition is followed by a correct or an incorrect one. A neighbourhood represents a choice in the LTS between branching behaviours that directly affect the property satisfaction, and it consists of incoming and outgoing transitions of that state.

Definition 12. *(Neighbourhood) Given an LTS* $M_E^k = (S_E, s_E^0, \Sigma_E, T_E)$ *where transitions have been typed as correct, incorrect or neutral, a state* $s \in S_E$ *where* $\forall t = s' \xrightarrow{l} s \in T_E$, t *is a neutral transition and* $\exists t' = s \xrightarrow{l} s'' \in T_E$, t' *is a correct or an incorrect transition, the neighbourhood of state* s *is the set of transitions* $T_{nb} \subseteq T_E$ *such that for each* $t'' \in T_{nb}$, *either* $t'' = s' \xrightarrow{l} s \in T_E$ *or* $t'' = s \xrightarrow{l} s''' \in T_E$.

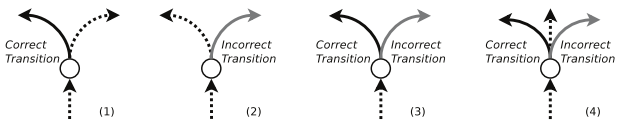

Fig. 3. The four types of neighbourhoods.

We can identify four different types of neighbourhoods by looking at their outgoing transitions (see Figure 3 from left to right). The first type consists of neighbourhoods with at least one correct transition and no incorrect transitions, and highlights a choice that can lead to behaviours that always comply with the sequence of inevitable actions. The second type is represented by neighbourhoods with at least one incorrect transition but no correct transitions, and highlights a choice that can lead to behaviours that never comply with the sequence of inevitable actions. The third and the fourth types have both at least one correct

and one incorrect outgoing transition, and they differ because of the presence (or not) of one (or more) neutral outgoing transition(s).

In Figure 4 we show the example described in Section 3.1 (in Figure 1), where the transition detection has allowed to detect neighbourhoods (states coloured in grey). In particular state 9, where correct and neutral transitions are present, shows a neighbourhood of the first type, where a choice that will always satisfy the property is possible. On the contrary, states 1, 6 and 8 show neighbourhoods of the second type, where a choice that leads to an incorrect behaviour is possible.

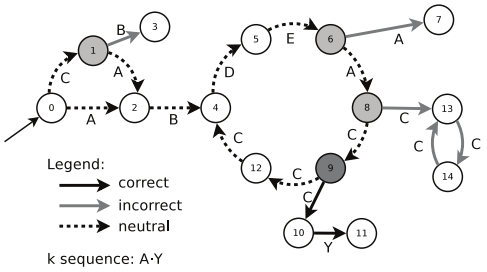

Fig. 4. Transitions classification and neighbourhoods

3.4 Simplification Techniques

The final step of our approach aims at simplifying the counterexample produced from the LTS and a given property. To do so we make a joint analysis of the counterexample and of the LTS enriched with neighbourhoods computed previously. This analysis can be used for obtaining different kinds of simplifications, such as: (i) an *abstracted counterexample*, that allows to remove from a counterexample actions that do not belong to neighbourhoods (and thus represent noise); (ii) a *shortest path to a neighbourhood*, which retrieves the shortest sequence of actions that leads to a neighbourhood; (iii) improved versions of (i) and (ii), where the user provides a pattern representing a sequence of non-contiguous actions, in order to allow the developer to focus on a specific part of the model; (iv) techniques focusing on a notion of *distance* to the bug in terms of neighbourhoods. For the sake of space, we focus on the abstracted counterexample in this paper.

Abstracted Counterexample. This technique abstracts a counterexample keeping only transitions that belong to neighbourhoods. The technique takes as input an LTS M where neighbourhoods have been identified, and a counterexample c, produced from M and from the inevitability property expressed by k. The procedure for the abstracted counterexample first identifies in c the states that belong to a neighbourhood in M. Second, it removes all the actions in c that do not represent incoming or outgoing transitions of neighbourhoods identified in the previous step. Figure 5 shows an example of a counterexample where two

neighbourhoods, highlighted on the right side, have been detected and allow us to identify actions that are preserved in the abstracted counterexample.

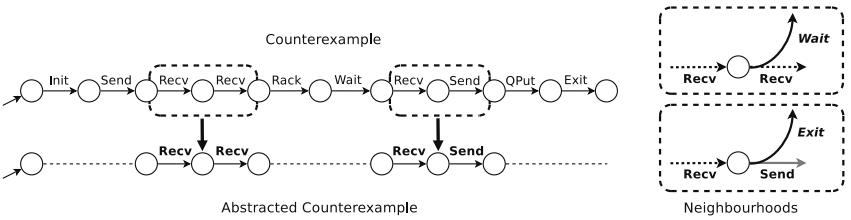

Fig. 5. Abstracted counterexample

4 Illustration on Case Studies

We implemented our approach in a Java tool, which consists of about 6000 lines of code. Compilers provided by the CADP toolbox [10] have been used to transform LNT [6] and LOTOS [5] specifications into LTS models, which are used as input format to our application. Our tool was applied to several examples in order to validate it. We present here two of them, showing the verification of both a simple and a nested inevitable execution property.

Sanitary Agency. We describe here the *Sanitary Agency* [16] case study, which models an agency that aims at supporting elderly citizens in receiving sanitary assistance from the public administration. The model is composed of four participants, depicted in Figure 6: a bank to manage fees and payments; a cooperative to provide welfare services; a citizen to perform the service requests; a sanitary agency to manage citizens' requests and which also contribute to the payment. For illustration purposes, we defined a property with two nested inevitable executions. The property states that the treatment of a citizen request by the agency (represented by the `REQ_EM` action) should always take place, and should always be followed by the reception of a transport service by the citizen (represented by the `PROVT_REC` action).

Our tool identified five neighbourhoods in the model. We then applied the abstracted counterexample technique to the shortest counterexample, allowing to discover two neighbourhoods and consequently reducing the length of the counterexample from 15 actions to 4. The top side of Figure 7 depicts the shortest counterexample while the bottom side depicts the corresponding neighbourhoods. The extracted actions are relevant since the neighbourhoods to which they belong precisely identify choices in the model that violate the property. In this case, the first neighbourhood shows that the first action in the property is not inevitable. The `REQ_EM` action can take place only after an `ACCEPTANCE_EM` action, but the neighbourhood exhibits an incorrect transition with the `REFUSAL_EM` action, revealing that the citizen request can be refused and thus preventing its

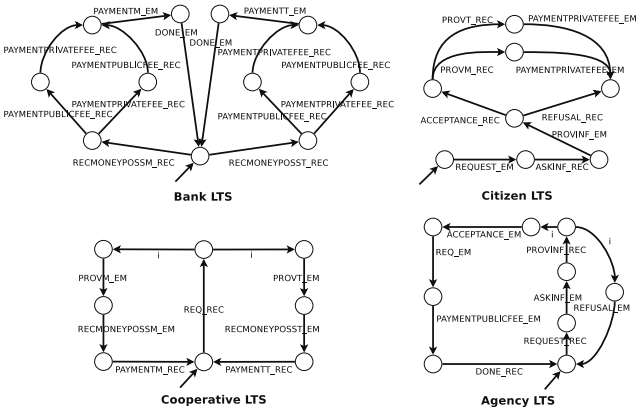

Fig. 6. Sanitary agency models

treatment. The second neighbourhood shows that, even when the citizen request is treated by the agency, the system does not always satisfy the nested inevitable action, since it can also provide meal services. This is highlighted by the choice between the correct transition with the `PROVT_EM` action (emission of a transport service) and the incorrect transition with the `PROVM_EM` action (emission of a meal service).

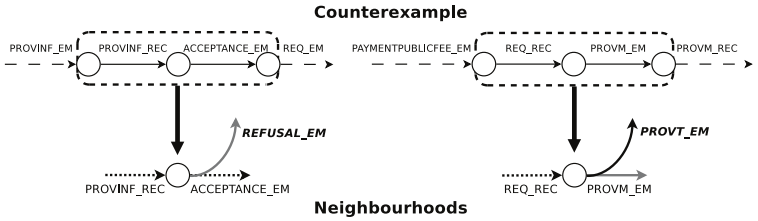

Fig. 7. Sanitary agency: shortest counterexample and neighbourhoods

Alternating Bit Protocol. We now discuss the *Alternating Bit Protocol* case study, which consists of a data link layer network protocol that allows the retransmission of lost or corrupted messages. The version of the protocol analysed here, available as CADP demo [12], is a variant without data values written in LOTOS. The model is composed of four processes: a transmitter process that acquires and sends a message; a receiver process that gets a message; medium1 and medium2 processes that represent transmission channels.

The demo is provided with an inevitable execution property that states that a `PUT` action will be eventually reached from the initial state. This property is not satisfied by the model because of the presence of loops in the specification that

can lead to an infinite trace that never reaches a `PUT` action. More precisely, the problem is caused by an interaction between the receiver and the medium2 processes. When the transmitter process has not yet started the message treatment (represented by the `PUT` action), the receiver might have to wait. In this case the receiver produces a `TIMEOUT` action, followed by the sending of an incorrect ack message (`RACK1` action). If this ack message is lost by medium2 (`LOSS` action) and the receiver is still waiting, a loop might be produced until the transmitter starts the message treatment.

Our tool detects six neighbourhoods in the model, all with correct and neutral transitions. Figure 8 depicts a portion of the model with these neighbourhoods, which are located at states 0, 2, 5, 12, 13 and 23. In particular, neighbourhoods at states 5, 12, 13, 23 present choices that make the execution of the system remain infinitely inside the loops, preventing the satisfaction of the property by reaching the `PUT` action. This is highlighted by neutral transitions containing `LOSS`, `RACK1` and `TIMEOUT` actions that repeat the cycle.

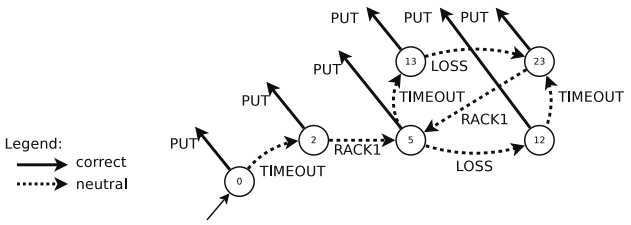

Fig. 8. Excerpt of the Alternating Bit protocol LTS model

5 Related Work

Causality analysis aims at relating causes and effects, which can help in debugging faults in (possibly concurrent) systems. This analysis relies on a notion of counterfactual reasoning, where alternative executions of the system are derived by assuming changes in the program. In [11], the authors present a general approach for causality analysis of system failures based on component specifications and observed component traces. In [3], the authors choose the Halpern and Pearl model to define causality checking in order to localise errors in hardware systems by analysing counterexample traces. Our approach is complementary to causality analysis since it helps to detect any kind of bugs and not only those involving causality.

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. CloSpan is also adopted in [14], where the authors apply sequential pattern mining to traces of counterexamples, in order to reveal unforeseen interleavings that may be a source of error. However,

reasoning on traces 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.

In [13] 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 different layers (representing distances from the bug) and produces a counterexample annotated with segments by exploring the model. Both our work and [13] 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.

Fault localisation for program debugging has been an active topic of research for many years in the software engineering community [18]. One of the main approaches in that line of work aims at localising faults using testing approaches. As an example, the approach presented in [15] relies on mutation testing to locate effectively the faulty statements. Experiments carried out in [15] reveal that mutation-based fault localisation is significantly more effective than current state-of-the-art fault localisation techniques. Note that this work applies on sequential C programs whereas we focus on formal models of concurrent programs.

We published in [2] an approach for counterexample analysis of safety property violation. [2] describes a preliminary version of neighbourhood and of the counterexample abstraction. The algorithmic solution using prefix/suffix annotations presented in Section 3 as well as the tool support and experiments presented in Section 4 are entirely new.

6 Conclusion

In this paper, we have proposed a method for improving the comprehension of counterexamples returned by a model checker when an inevitability property is not satisfied on a given behavioural model. To do so, we have first defined an algorithm to enrich the LTS that represents the model of the system with notions of prefixes and suffixes, which express parts of the sequence of inevitable actions. Second, we have provided a method to extract relevant portions of the LTS, called neighbourhoods, which highlight choices between a correct and an incorrect behaviour. Third, we have proposed a set of simplification techniques to extract relevant information that explains the cause of the bug, exploiting the notion of neighbourhood. All the steps of our approach are automated by a tool we implemented. The resulting simplified counterexample gives an improved explanation of the bug, as we have shown on experiments we carried out on real-world examples.

Acknowledgements. We are grateful to Radu Mateescu for his valuable inputs on liveness properties.

References

1. C. Baier and J. Katoen. *Principles of Model Checking*. MIT Press, 2008.
2. G. Barbon, V. Leroy, and G. Salaün. Debugging of Concurrent Systems Using Counterexample Analysis. In *Proc. of FSEN'17*, volume 10522 of *LNCS*, pages 20–34. Springer, 2017.
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*, pages 203–221. 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*, pages 162–177. Springer, 2014.
5. T. Bolognesi and E. Brinksma. Introduction to the ISO Specification Language LOTOS. *Computer Networks*, 14:25–59, 1987.
6. D. Champelovier, X. Clerc, H. Garavel, Y. Guerte, F. Lang, C. McKinty, V. Powazny, W. Serwe, and G. Smeding. Reference Manual of the LNT to LOTOS Translator (Version 6.7). INRIA/VASY and INRIA/CONVECS, 153 pages, 2018.
7. E. M. Clarke, S. Jha, Y. Lu, and H. Veith. Tree-Like Counterexamples in Model Checking. In *Proc. of LICS'02*, pages 19–29. IEEE Computer Society, 2002.
8. R. De Nicola and F. W. Vaandrager. Action versus State based Logics for Transition Systems. In *Proc. of Semantics of Systems of Concurrent Processes, 1990*, volume 469 of *LNCS*, pages 407–419. Springer, 1990.
9. M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in Property Specifications for Finite-State Verification. In *Proc. of ICSE'99*, pages 411–420. ACM, 1999.
10. 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.
11. G. Gößler and D. L. Métayer. A General Trace-Based Framework of Logical Causality. In *Proc. of FACS'13*, volume 8348 of *LNCS*, pages 157–173. Springer, 2013.
12. Inria CONVECS team. CADP demo 01: Alternating Bit Protocol. http://cadp.inria.fr/demos.html.
13. H. Jin, K. Ravi, and F. Somenzi. Fate and Free Will in Error Traces. In *Proc. of TACAS'02*, volume 2280 of *LNCS*, pages 445–459. Springer, 2002.
14. S. Leue and M. T. Befrouei. Mining Sequential Patterns to Explain Concurrent Counterexamples. In *Proc. of SPIN'13*, volume 7976 of *LNCS*, pages 264–281. Springer, 2013.
15. M. Papadakis and Y. L. Traon. Effective Fault Localization via Mutation Analysis: A Selective Mutation Approach. In *Proc. of SAC'14*, pages 1293–1300. ACM, 2014.
16. G. Salaün, L. Bordeaux, and M. Schaerf. Describing and Reasoning on Web Services using Process Algebra. In *Proc. of ICWS'04*, pages 43–50. IEEE Computer Society, 2004.
17. R. E. Tarjan. Depth-First Search and Linear Graph Algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.
18. W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa. A Survey on Software Fault Localization. *IEEE Trans. Software Eng.*, 42(8):707–740, 2016.